NADOL™

Nibbles Away
Disk Optimized
Language

written by
Randy Ubillos

User's Manual

N.A.D.O.L.

Nibbles Away Disk Optimized Language

version 1.2

Correction to the NADOL Manual

Along with the enclosed replacement pages for the NADOL manual, the following corrections should be made to your manual:

Information from Chapter's 11 and 12 have been incorporated into Chapter 10.
their references should be removed from the table of contents.

On page 3.1, in the last paragraph, the phrase "and then add those two results together" should read "and then multiply those two results".

On page 5.3, the result from the program should print the numbers from 0 to 4, not 0 to 5.

On page 5.4, the program should start with the statement "DEFINE INTEGER I,J,K" instead of the other way around.

On page 5.6, the second program should have "WHILE 1" as its loop control statement, since this forces ther condition to always be true.

On page 8.49A, under syntax, the statement is called "RSYNC" not "WSYNC".

On page 10.2, the last paragraph should have the following sentence in it: "NADOL is shipped to use slot 1 for a printer...".

8-9-85

From the Editor's Desk:

COMPUTER:applications, Inc. introduces N.A.D.O.L.,™ our interpreted language for the Apple II line of computers. We believe that by using the NADOL language, you will be provided with a state-of the art tool for designing utility software, and other highly interactive applications.

Nibbles Away III has been included with the NADOL package as an example of the power and versatility of our new language. The Nibbles Away III section will be enhanced over time with a Program Look-up Database, powerful editing utilities, and sophisticated backup modules.

Before you begin to explore the features of NADOL, this manual needs to be put in order. There is a set of addendums, and chapter 10 which should be inserted into the manual. Also, the section divider tabs should be inserted. The 'NADOL' tab goes in front of chapter 1, 'NAIII AND OTHER UTILITIES' goes in front of chapter 10, 'DISK PROTECTION' goes in front of chapter 13, and 'APPENDIXES' goes in front of the appendixes.

Please take a moment to make a backup of your original NADOL diskette. An extra label is provided for this purpose. NADOL is not copy protected, so you may make as many copies as you wish *for your own use only.* Please do not give away copies of this software.

For those who wish to keep up with the new utilities being developed in NADOL, or for those who want to know the latest 'happenings' regarding copy protection, the NIBBLE NEWS™ subscription newsletter service is available. NIBBLE NEWS is the COMPUTER:applications Inc. newsletter, dedicated to Nibbles Away parameters and NADOL programs. This newsletter contains Tutorials, helpful hints, and many useful and interesting programs. It is our way of bringing you the most recent developments as quickly as possible. If you wish to subscribe to Nibble News, please contact COMPUTER:applications, Inc. at (919) 846-1411.

I know that you are anxious to explore all of the inner details of this package, and we hope that you have as much fun using NADOL and NAIII as we did in making it.

Randy Ubillos

Important!

You should make a backup copy of your NADOL master disk NOW. This can be done by using the 'FAST SECTOR COPY' option from the NAIII main menu (option #2). After making a backup, put your original disk away. A second disk label is supplied to be placed on this backup disk.

Well, here it is. The long awaited Nibbles Away III. As you have probably noticed, NAIII is actually a program written in NADOL™, our new interpreted language for the Apple computer. We believe you will find that it provides the state-of-the-art in software backup, and goes well beyond in its capabilities for developing highly interactive disk utilities.

The first step is to arrange your manual. There is a set of addendums, and chapter 10 which should be inserted into the manual. Also, the cardboard dividers should be inserted. 'NADOL' goes in front of chapter 1, 'NAIII and other utilities' goes in front of chapter 10, 'Disk Protection' goes in front of chapter 13, and 'Appendixes' goes in front of the appendixes.

Please take a moment to make a backup of your original disk. An extra label is provided for this purpose. NADOL is not copy protected, so you may make as many copies as you wish *for your own use only*. Please do not give away copies of this software.

If you wish to keep up with "what's happening" with NADOL, current lists of parameters, and many useful and interesting programs for NADOL, 'Nibble News' is available. Nibble News is the COMPUTER:applications Inc. newsletter, dedicated to Nibbles Away (and now NADOL too!). It is our way to bring you the most recent developments as quickly as possible. If you are interested in subscribing to Nibble News, please contact COMPUTER:applications, Inc. at (919) 846-1411.

I know that you are anxious to explore all of the inner details of this package, and we hope that you have as much fun using NADOL and NAIII as we did in making it the leader in software backup technology.

RANDY HILLOS

P.S. In case anyone was wondering, this entire manual, pictures and all, was done 100% on an Apple Macintosh with an imagewriter printer. Every page was produced camera ready, with no need for physical 'cut and paste' editing!

DISCLAIMER OF ALL WARRANTY AND LIABILITY

COMPUTER:applications Inc., or any dealer distributing this product, makes NO WARRANTY, either EXPRESS or IMPLIED, with respect to the information provided, or to the floppy diskette, its quality, performance, merchantability, or fitness for any particular use. It is solely the purchaser's responsibility to determine its suitability for any particular purpose.

COMPUTER:applications, Inc. will in no event be held liable for direct, indirect, or incidental damages resulting from any defect or ommision in the information provided, the floppy diskette, or other processes including, but not limited to any interruption of service, loss of business or anticipatory profit, legal action, or other consequential damages.

THE USER ASSUMES ALL RESPONSIBILITY ARISING FROM THE USE OF THIS SOFTWARE!

COMPUTER:applications, Inc. reserves the right to make corrections or improvements to the information provided, and to the related software at any time without notice.

COMPUTER:applications, Inc.
12813 Lindley Drive
Raleigh, NC 27609
(919) 846-1411

Table of Contents

SECTION 1:

A History of Nibbles Away ..... Chapter 1
Using NADOL ..... Chapter 2
Built-in Math ..... Chapter 3
Using the Editor ..... Chapter 4
NADOL tutorial ..... Chapter 5
Procedures and Functions ..... Chapter 6
Examples to try ..... Chapter 7
Built-in Statements ..... Chapter 8
Built-in Variables ..... Chapter 9

SECTION 2:

Nibbles Away III ..... Chapter 10
Configure ..... Chapter 11
16 sector copy ..... Chapter 12

SECTION 3:

Disk Protection ..... Chapter 13

SECTION 4:

Error messages ..... Appendix A
Decimal/Hex/ASCII chart ..... Appendix B
Disk Speed Adjustment ..... Appendix C
NADOL memory map ..... Appendix D
Quick reference chart ..... Appendix E

Chapter 1

A History of Nibbles Away

A HISTORY OF NIBBLES AWAY

Nibbles Away began back in 1980 as an experiment into the disk protection schemes which had begun to come into use. Prior to that time most disks were directly copyable, and back-ups were as simple as using the Apple-copy program supplied on the master disk.

The software companies claimed that the protection was to protect their interests against software pirates. While no one tries to pretend that software pirates do not exist, the fact that legal owners of a program need to make back-ups to protect their software investment, is largely disregarded by the software companies.

Most companies would repair a 'blown' disk if the user was willing to part with the disk for a week or two while it was shipped around the country with their valuable payroll or database information. Obviously this was not a reasonable solution.

Even so, it seemed that as soon as one disk appeared using a protection system, many more began to use similar systems until the disks without protection were few and far between.

In 1981 Nibbles Away version A1 was announced as a disk backup program to allow software users to protect their investment.

The software companies quickly found that the flexibility of the Apple disk drive hardware allowed an incredible variety of protection schemes to be implemented. To combat these new techniques, in early 1982 Nibbles Away II was announced.

Nibbles Away II had a much more flexible disk back-up system, easy to use and modify parameters with names instead of numbers. Also included was a full set of track and sector editors which allowed the user to view the data on a disk directly to determine the type of protection being used.

But still the software companies pushed forward to more and more extensive protection systems, many times contracting companies whose only programming service was protecting diskettes. Many times these protection systems cost a great deal of money, which resulted in higher software costs. Some of the schemes were unreliable, in some cases with failure rates as high as 50% for the finished diskettes.

With the incredibly diverse selection of protection systems on the market, each having a large number of variations within itself, the total number of protection systems is staggering and extremely difficult for a single program to keep up with.

Taking all of this into consideration, COMPUTER:applications decided that a giant step forward was needed to provided users with a comprehensive program capable of keeping up with the ever changing protection systems.

The concept of NIBBLES AWAY III was developed to provide the maximum possible flexibility.

At this point one item should be clarified. The menus and prompts that are seen on the screen during a normal copy are actually done through a program written in NADOL, the Nibbles Away Disk Optimized Language. This program can be modified by the user at will. This allows NAIII to stay current, no matter what developments take place.

An added benefit is that users can create their own programs under this powerful language. A few examples would be programs which can convert DOS 3.3 files to Apple CP/M, or one which could display a color disk map of the files on a diskette. The possibilities are almost limitless.

Users should not be scared off by the term 'language'. NADOL is a language in the same way that Applesoft is a language. Anyone who can program in Applesoft can program in NADOL. In many cases it is much easier, especially for operations involving diskettes, since many of NADOL's functions make handling disk information easy.

Chapter 2

Using NADOL

USING NADOL

NADOL is an interactive language which can be used in two distinct modes. The first is *immediate* mode, where commands are executed as they are typed in. The second is *deferred* mode, where a series of commands is typed in and executed at a later time.

To become aquainted with NADOL, immediate mode is the best thing to start with. In this section we will go through several examples. The way to get the most out of this section is to have NADOL up and running on the screen and try the examples as they are given. This type of 'hands on' training usually gives the best results.

To enter NADOL itself, choose the NADOL option from the main Nibbles Away III menu. This will display a period '.', which is the prompt for the NADOL immediate mode.

At this point, try a simple command. Type:

```
PRINT "HELLO"
```

And press the <RETURN> key. The word HELLO will be printed on the screen below the statement which was typed in. Numeric expressions work in the same manner. Type in the following:

```
PRINT 5*6
```

This will print 30 on the screen. Most of the commands which are listed in chapter 8 can be typed in directly to give immediate results. Some procedures require variables to be passed as their parameters. To do this the variables must first be defined. As an example, the following sequence of commands will read in and display the data from sector 5 on track 4:

```
DEFINE INTEGER TRACK,SECTOR,COUNT,ERR
TRACK=4
SECTOR=5
COUNT=1
RSECT(RBUF,TRACK,0,SECTOR,COUNT,6,1,ERR)
DISPLAY(RBUF,256)
```

The **DEFINE** statement creates the four variables which are needed to pass to the **RSECT** routine. Then the track and sector numbers are assigned to the variables with the '=' operator. The variable **COUNT** is set to 1 since only one sector is to be read. The **RSECT** procedure is called to perform the actual read operation. Then the **DISPLAY** procedure is called to show the data which was just read in. The predefined variable **RBUF** is a section of memory normally used for raw data reads, but it can be used for any other purpose as well. In this case we used it as a temporary storage buffer for the data that was read and displayed.

At any time that NADOL is waiting for a response (when there is a blinking cursor present on the screen), the ctrl-P keystroke may be used to print the contents of the screen. The predefined variable **PRTSLOT** is used to control which slot the data will be sent to. It is normally set to 1, since that is the standard slot for a printer interface card. Any slot from 1 to 7 may be used if a proper interface card exists in that slot. (In writing this manual, the screen dumps were embedded in the text by setting **PRTSLOT** to the slot of a serial interface card connected to the word processing computer, allowing direct transfers of screen images in to the text seen here). The ctrl-P command is very useful for saving data for future reference.

☞ Be sure that the output device is ready before using ctrl-P.

In the example above we saw how integer variables were created. Integer variables are the normal type used for most operations. They can contain numbers from -32767 to 32768. For some operations it is desirable to use byte variables. Byte variables can contain values from 0 to 255. The main difference is that integers occupy two bytes of memory, while bytes occupy, what else, one byte.

There are many instances where it is desirable to access many variables as a unit. This might be the case where a sector of data is read in off a diskette. Instead of assigning each byte to a separate variable, which would make getting to the information very difficult, NADOL provides arrays. Arrays are groupings of variables. They are all referred to by the same name, with the *subscript* distinguishing between them. Below is an example of how to create and use an integer array:

```
DEFINE INTEGER[10] MINE
MINE[0]=1
MINE[4]=5
PRINT MINE[0]+MINE[4]
```

This would print **6**, since the two values added together were 1 and 5. Each separate variable is called an *element* of the array. The number between the brackets, which determines which element is being referred to, is called the *subscript*. The subscript may be either a number, or it may be a variable or expression. See Chapter 3 for more information on expressions.

At this point it should be noted that not all Apple keyboards can normally generate the left bracket. On the Apple //e and //c both brackets are present on the keyboard. On the Apple][and][+ the right bracket can be entered with the shift-M key, and within NADOL the left bracket may be entered using the shift-N key.

When an array is defined, the highest desired element is specified in the **DEFINE** statement. If the number 10 is used, as in the example above, subscripts may be in the range 0 through 10, giving 11 elements total. There is actually no checking performed by NADOL to verify that the subscript used with an array is within the limits of its definition. It is up to the user to make sure that the subscript is within the range originally defined, or unpredictable results may occur.

Any variables which are defined take space away from NADOL's free space. Both programs and variables use up free space. Care should be taken not to allocate too many variables if a large program is to be entered. Many times the predefined buffers **RBUF** and **WBUF** can be used for temporary data storage instead of declaring special buffers for a particular application.

All data which is managed by NADOL is stored in either bytes or integers, or arrays of one or the other. Many times it is desired to store text data in a program. NADOL has provisions for handling this built in. Text is stored in byte arrays in a special format. The text starts at element 0 of the array, with each additional character following in sequential locations of the array. The element following the end of the text should contain a value of zero. This signifies the end of the string to NADOL.

Normally when an array variable is specified in a **PRINT** statement, the value of the first element (or whichever one was specified by a subscript) is displayed. In order to print the text contained in a byte array, an exclamation point should be placed in front of its name in the **PRINT** statement. Take the following example:

```
DEFINE BYTE[30] STRING
STRING[0]="H"
STRING[1]="I"
STRING[2]=0
PRINT !STRING
```

Would print **HI** on the screen. (The **PACK** procedure, detailed in chapter 8, is the normal way to put text into byte arrays). This shows how a byte array can be used as a string in a program.

In order to execute programs in deferred mode, they must be typed in with the built-in editor. Chapter 4 explains the use of the editor, while chapter 5 describes programming in the NADOL deferred mode.

Chapter 3

Built-in Math

BUILT-IN MATH

Mathematical operations make up a large part of what a program is usually required to do. NADOL has a large number of built-in math functions to allow the programmer a great deal of freedom when writing a program.

Since NADOL is designed as a utility language, integers are the most frequently used numbers. NADOL uses '16-bit' signed integers for all of its internal calculations. This results in a range of -32767 to 32768 for all numerical values which are handled by NADOL.

For performing most calculations, the standard mathematical operators are provided. Along with these are several logical operators (described below) and several comparison operators (also described below).

When NADOL looks at an expression and begins to evaluate it, there is a set of rules which determines which operations are performed first, and which are performed last. This is called *precedence*. Those operators with the highest precedence will be evaluated first. The list of available operators, from highest to lowest precedence, is as follows:

* /	Multiplication and division
+ -	Addition and subtraction
<> = <= >= AND OR XOR	Comparisons and logical operators

This order of precedence means that the expression:

```
5*6+3*4
```

Will result in a value of 42. NADOL first evaluates the two multiplications, since they have the highest precedence. Then the addition is performed. Sometimes it will be desirable to overide the order of precedence. Take the following example:L

```
7+3*6+9
```

If the actual intent of this expression was to add 7 and 3, then add 9 and 6, and then add those two results together, the result would not be the desired value, since NADOL will perform 3*6 first, and then add the 7 and the 9. To avoid this, parentheses may be used, as in the following example:

```
(7+3)*(6+9)
```

The parentheses inform NADOL to evaluate everything inside the parentheses first, and then to use the results in the multiplication. The expressions within the parentheses may be any standard expression, and may even include nested parentheses. Remember that within each set of parentheses, NADOL follows the same rules of precedence to evaluate what it finds.

The logical operators are AND, OR, and XOR. These provide the logical AND, OR, and Exclusive-OR operations, respectively. Each of these operators performs their corresponding logical test on each bit of the two sixteen bit operands, and returns a sixteen bit result.

The comparison operators return a 1 or 0 result depending on whether or not the condition associated with the operator is true or false, based on the two arguments. These operators are useful when used in conjunction with the flow control statements (see chapter 5).

When entering numbers, NADOL assumes that they are decimal, unless a dollar sign ($) is placed in front of the number, which specifies base sixteen, hexadecimal. This means that '10' would be evaluated as a decimal ten, while '$10' would be the hexadecimal representation for the number sixteen. These two representations may be mixed freely in expressions. Internally all values are stored in hexadecimal. Before any assignments or comparisons are performed, all operators are converted to hexadecimal, so the user need not be concerned about comparing hexadecimal and decimal values.

Many times it is desirable to be able to use the ASCII value of a particular character in a comparison or expression. NADOL allows this by placing double quotes (") around the desired character. For example, "A" would evaluate to 65, which is the hexadecimal equivalent for the letter A.

Chapter 4

Using the Editor

THE EDITOR

NADOL has a built-in program editor for creating or modifying programs. This editor allows the user to scan through a program, insert and delete lines or characters, and make changes at any point in the program. It is a highly interactive editor with a set of control-key driven commands.

The editor can handle program lines up to 250 characters in length, and supports horizontal (sideways) scrolling to allow these long lines to be displayed, no matter which version of the editor is being used.

There are several different versions of the editor included on the NADOL master disk. The standard editor is the 40 column version. Versions are also included for the Apple //e, Apple //c, Videx, Sup'r'term and Smarterm 80 column cards.

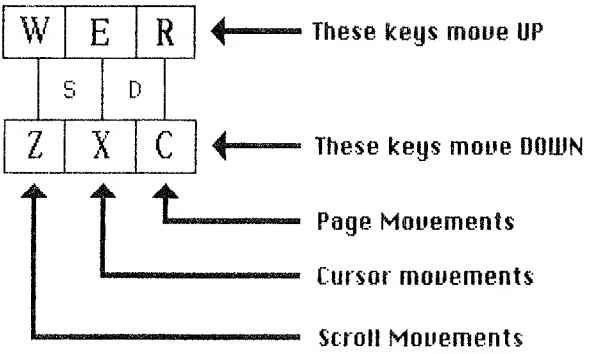
The editor is invoked by typing EDIT from the '.' prompt of NADOL. This will display any current program and place the cursor at the top of the screen. If an 80 column version of the editor is being used, the card will be activated at this time. If the 80 column card does not have a built in video switch, the user should switch cables whenever the editor is activated.

If no program is in memory, the cursor will be at the top of the screen with [END OF TEXT] displayed. This shows that there are no lines of program currently stored. If a program is currently in memory when the editor is invoked, it will be displayed on the screen. If it is desired to enter a new program, then the "NEW" command should be typed before entering the editor.

Due to the larger screen area of an 80 column card, the speed of the editor is somewhat slower when using one of these versions. Some users may wish to stay with the 40 column editor due to its greater speed, and since it can perform horizontal scrolling to view programs of any width.

The commands of the editor, and their descriptions are listed below. The six cursor movement keys have been chosen for their physical location on the keyboard.

The keys 'W','E', and 'R' all perform upward movement, and the keys 'Z','X' and 'C' all perform downward movement, each one being the opposite of the one above it. The layout of these keys makes moving the cursor very easy.



The full list of commands is as follows:

Keystroke	Result
left arrow	Moves the cursor to the left one space at a time.
right arrow	Moves the cursor to the right one space at a time.
ctrl-A	Selects Add mode. All subsequent alphanumeric keys will be inserted into the current line at the cursor position.
ctrl-B	Moves the cursor to the Beginning of the current line.
ctrl-C	Page down, described above.
ctrl-D	Deletes the character to the right of the cursor on the current line.
ctrl-E	Cursor up, described above.

ctrl-F	Restores previous contents of line. Used to undo any changes made since the cursor was placed on this line.
ctrl-G	Moves the cursor to the last line which was executed from NADOL. This is useful for finding the line where an error occurred from within a program.
ctrl-I	Tabs to the next tab stop. Tab stops are set every 2 character positions.
ctrl-L	Inserts a new Line at the current cursor position. All subsequent lines are moved down one line.
ctrl-N	Moves to the End of the current line.
ctrl-Q	Quits the editor and returns to NADOL.
ctrl-R	Page up, as described above.
ctrl-W	Scroll up, described above.
ctrl-X	Cursor down, described above.
ctrl-Y	Deletes the line which the cursor currently occupies. All subsequent lines are moved up one line.
ctrl-Z	Scroll down, described above.

These commands may be used as many times as desired, in any order.

Alphanumeric characters are placed at the cursor location as they are typed, replacing any characters which may have previously existed under the cursor. The cursor moves one position to the right for each character which is typed in.

If the cursor moves onto the END OF TEXT marker, a new line will be inserted at that point, allowing program to be entered without issuing the ctrl-L command.

Chapter 5

NADOL Tutorial

THE LANGUAGE

NADOL is a structured programming language borrowing from the strong points of several popular languages such as Pascal, BASIC and C. This blend gives a language in which it is easy to write programs, and is able to perform a wide range of tasks with ease.

Programs are arranged as one statement per line, one executing after another. Several branching commands are available to control the flow of the program based on conditions set up by the user.

The best way to start is with an example; so we will pick a simple program to print out a number.

The editor should be used to type in these programs. See chapter 4 for a description of how to use the editor.

Type in the following program:

```
PRINT 6*7
```

Now exit the editor (ctrl-Q) and type 'RUN'. The following should be printed:

```
42
```

Typing 'RUN' caused NADOL to look at the program which had been entered, starting at the first line. In this case, there was only one line; a simple print statement. If there had been more lines, NADOL would have performed the task specified by each one in the order in which they appeared in the program.

As NADOL steps through each line of a user program, it has several options based on what it finds. Each line in a program can take on one of four basic types:

1. Statement.
2. Assignment.
3. Procedure call.
4. Flow control.

Statements are lines which specify an action to take place, such as initializing a disk, or setting a screen mode. These lines cause an action to take place and then execution continues with the next line. An example would be "**PRINT (5+6)/3**".

Assignments cause the result of an expression to be assigned to a variable. An example would be "**I=5**".

Procedure Calls take a certain number of parameters from the line and pass them to the specified routine. An example of this would be "**DISPLAY($800,40)**". These lines cause an action to take place and then execution continues with the next line.

Flow Control lines cause a change in the normal flow of execution of statements. These lines include IF/ELSE/ENDIF, WHILE/ENDWHILE and the infamous GOTO. These are the lines which make NADOL programable, and those which will be discussed in detail in this chapter.

Flow control statements allow a program to execute a series of commands over and over again, or to execute certain sections of commands only when predefined conditions arise.

The WHILE/ENDWHILE statements cause a series of commands to be executed until a condition is met. The best way to see this is with an example (Be sure to erase any previous program which was entered, by typing "NEW"). Key in the program on the next page.

```
DEFINE INTEGER ME
ME=0
WHILE ME<5
   PRINT ME
   ME=ME+1
ENDWHILE
PRINT "DONE"
```

Now try running the program by typing "RUN". The screen should look like this:

```
0
1
2
3
4
5
DONE
```

First, an explanation of some of the statements used is in order (see chapter 8 for a full description of all statements). The **DEFINE** statement created a variable called **ME** to which we can assign values, and which we can perform tests on. The assignment **ME=0** set the value of **ME** to zero.

The **WHILE** statement creates a loop. Everything between the **WHILE** and **ENDWHILE** statements is executed until **ME<5** is no longer true. Inside the loop we simply print **ME** and then add 1 to the value of **ME**.

The best way to understand what happens during a WHILE/ENDWHILE loop is to follow the same steps that NADOL follows when it executes the program:

The DEFINE statement creates a variable, and the following assignment sets its value to zero. The first time that we reach the **WHILE** statement, **ME** is zero, so **ME** is less than 5, and the print and increment statements are executed. The second time, **ME** is one, and so on until **ME** reaches 5, when it is no longer less than 5. At this point the loop exits, and the program continues at the statement following the **ENDWHILE** statement, stopping after the last line in the program.

The section of the program between the WHILE and ENDWHILE lines is called the WHILE/ENDWHILE *block*. Within this block, other WHILE/ENDWHILE blocks may exist. This is called *nesting*. WHILE/ENDWHILE blocks may be nested up to 8 levels deep. An example of a program which is nested 3 levels deep is shown below:

```
DEFINE I,J,K INTEGER
I=1
WHILE I<=3
  J=1
  WHILE J<=2
    K=1
    WHILE K<=8
      PRINT "*";
      K=K+1
    ENDWHILE
    PRINT
    J=J+1
  ENDWHILE
  PRINT
  I=I+1
ENDWHILE
```

When run, this program would create the following display on the screen:

```
********
********

********
********

********
********
```

This shows how WHILE/ENDWHILE blocks may be nested within each other. For additional examples, examine the programs in chapter 7.

The IF/ELSE/ENDIF statements are the next topic. The statements cause a series of commands to be executed only if a specific condition is true. An example is in order:

```
IF HASLC>0
  PRINT "YOU HAVE A LANGUAGE CARD"
ENDIF
```

This short program will cause the message specified to be printed only if the value of the predefined variable HASLC is greater than zero (See chapter 9 for a list of predefined variables). Many times this form of the IF/ELSE/ENDIF statement is enough, but there are many times when one action should be taken when the specified condition is true, and another action should be taken when it is not true. Examine the following addition to the program above:

```
IF HASLC>0
  PRINT "YOU HAVE A LANGUAGE CARD"
ELSE
  PRINT "THERE IS NO LANGUAGE CARD IN THIS COMPUTER"
ENDIF
```

In this example, we have two messages which can print out, depending on whether the condition "HASLC>0" is true or false.

This shows how we can control which parts of a program actually are executed, depending on certain conditions. As with WHILE/ENDWHILE blocks, IF/ELSE/ENDIF blocks may be nested up to 8 levels deep. IF/ELSE/ENDIF and WHILE/ENDWHILE blocks may also be nested within each other, in any order, as long as the blocks do not 'cross'. An example of blocks which cross would be:

```
DEFINE INTEGER ATE,EAT
ATE=5
EAT=ATE*7
WHILE ATE>3
  ATE=ATE+1
  IF ATE>EAT
    ATE=ATE+3
ENDWHILE
  ENDIF
```

In this example, the two control blocks overlap each other, creating a program which makes very little sense. In all of the programs shown so far, a convention has been set up where the lines inside each block are indented by 2 spaces. This makes errors like the one in the program above stand out clearly. This convention should be followed in all programs to make debugging easier.

The final type of flow control statement is GOTO. The GOTO statement has a rather rocky past. Many proponents of structured programming have decided that it should be banned since, it can lead to "spaghetti bowl" programs. GOTO is provided in NADOL because there are certain cases where a single GOTO can reduce the overall size of a routine substantially. It is not intended to be the sole type of branching used within a program! To illustrate it's use, we must first talk about the LABEL statement. The LABEL statement defines a location which can be branched to using a GOTO. The following program gives an example of both of these:

```
DEFINE INTEGER I
I=1
LABEL LOOP
PRINT I
I=I+1
GOTO LOOP
```

This program will print a series of numbers starting at 1 and running up until the ctrl-C key has been pressed. This program, like most others, should be coded with a WHILE/ENDWHILE block to make it more readable. WHILE/ENDWHILE blocks which never terminate can be implemented using a control expression which is always zero, such as the number 0. This would result in the following program:

```
DEFINE INTEGER I
I=1
WHILE 0 1
  PRINT I
  I=I+1
ENDWHILE
```

This is easier to understand in a large program, and the GOTO statement should be avoided whenever possible to insure readability and ease of debugging later on.

As we have seen from this section, flow control statements are the backbone of NADOL. A language without the ability to change the order in which it steps through a program is little more than a complicated mimic, able only to perform a task in a specific order, leading to a single end result. Flow control statements, however, allow a program to perform different tasks based on decisions. This allows some operations to be repeated until a particular task is completed. Very sophisticated and 'intelligent' programs may be devised by making extensive use of the conditional facilities of NADOL. In all, the flow control statements, combined with the large selection of built-in routines within NADOL, make up a very powerful environment for almost any type of application.

This completes our description of the flow control statements of NADOL. Chapter 7 provides a number of examples showing how these statements are used in actual programs, and how they interact with other statements.

The NADOL master disk contains the programs in chapter 7 already entered into NADOL format. The best way to learn what the different statements do is to make changes to the programs and view the effect that the changes have on the operation of the program (Always be sure to try these changes on a BACKUP copy of the NADOL master disk, since some changes may not be as harmless as they seem).

Chapter 6

Procedures and Functions

PROCEDURES and FUNCTIONS

NADOL provides two different ways to create subroutines in a program. Which of the two is used for a particular application will depend on the situation involved. *Procedures* perform a set of instructions and return to the main program. *Functions* pass back a value to the main program.

The best way to illustrate how this works is with an example. The following program contains a short procedure:

```
DEFINE INTEGER J

PROCEDURE TIMESTWO
   J=J*2
ENDPROC

J=8
PRINT J
TIMESTWO
PRINT J
```

If this program is run, the numbers 8 and 16 will be printed. This is what happens:

The **DEFINE** statement creates the variable J.
A procedure is defined as the lines between the the **PROCEDURE** and **ENDPROC** statements.
J is set to 8 and printed.
The statement **TIMESTWO** in the main program transfers control to the procedure with that name.
TIMESTWO then multiplies J by 2 and exits.
J is then printed again, with its new value of 16 (8*2).

The procedure **TIMESTWO** might be useful if J were to be multiplied by 2 many times during a program. It would be more useful, in most cases, if the procedure could multiply any number by 2. This is where parameter passing comes into play.

When a procedure or function is called, it can be followed by an open parentheses, a list of variables or expressions, and a close parentheses. The variables or values within the parentheses are known as *parameters*.

These parameters then become available to the procedure or function. The difference is that different parameters can be passed from different points within a program.

Up to 8 parameters may be passed to a procedure or function at any one time. From within the procedure, they are referenced as **%1** through **%8**. The percent sign signifies that a passed parameter is being referenced.

Another example is in order. Let's make a new program with a more complicated procedure which will set the third passed parameter to 0 or 1, depending on whether the first two parameters are equal or not.

```
DEFINE INTEGER A,B,C,J

PROCEDURE COMPARE
   IF %1 = %2
      %3 = 1
   ELSE
      %3 = 0
   ENDIF
ENDPROC

COMPARE(1,2,J)
PRINT J
COMPARE (4,2+2,J)
PRINT J
```

In this case, 0 and then 1 would be printed. The procedure is defined to check the values of the first two passed parameters, **%1** and **%2**. If they are equal, as they are in the second case, the the third passed parameter, **%3**, is set to a one. If they are not equal, as in the first case, **%3** is set to 0. These are the two values displayed when this example is run.

Note here that the third passed parameter was a variable name, and that modifying the third parameter from inside the procedure modified the value of J in the main program. In this way, a procedure can pass information back to the routine which called it.

Be sure to specify a variable name for any parameter in which information is going to be returned from a procedure.

If more than one procedure is defined, one procedure can call another, up to eleven levels.

A procedure may pass its parameters to another procedure, in any position in the parameter list. The called procedure will then reference the parameter based on the position in *its* parameter list. Any changes made to the parameter will affect the parameter in the first subroutine, as well as the passed variable in the main program.

Note that while procedures and functions may occur at any point within a program, it is a good idea to place them at the beginning of the program.

Functions operate in much the same way as procedures, except they act as expressions rather than statements. Take the following example:

```
DEFINE INTEGER J

FUNCTION OURS
   %1 = %1 + 1
   RESULT= %1 * %1
ENDFUNC

PRINT OURS(7)
J = OURS(6)
PRINT J
```

As shown in this example, a function is used on the right hand side of an equate, and returns a value.

The **RESULT=** statement defines the value which will be passed back to the calling program. Passed parameters are handled in exactly the same way as for procedures.

Functions are very useful for subroutines which generate a single value. If more than one value is generated by the subroutine, a procedure may be a better way to perform the task, using the parameter system to return multiple values to the calling program.

The difference between procedures and functions may not be immediately apparent. An analogy would be to compare procedures/functions to messengers. Each one can carry up to eight packages, or *parameters*, in their pack when they are sent out. During their travels, the items in their packs may be modified. These modifications are retained when the messenger returns. The only difference is that a function also brings back an extra value. This analogy is illustrated in figure one below.

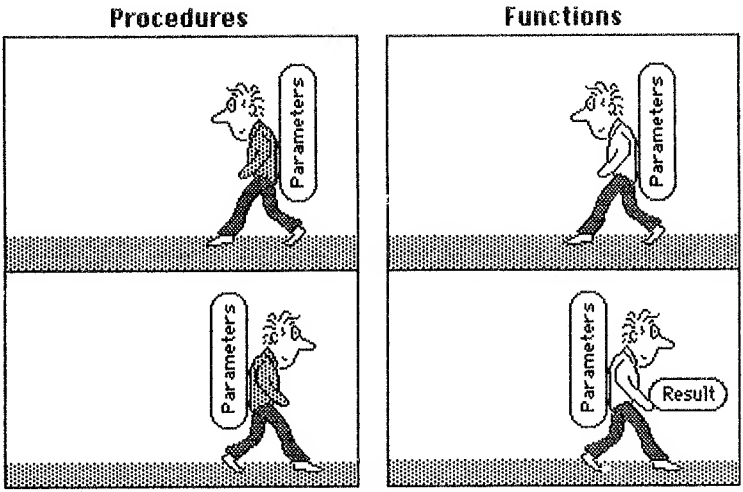
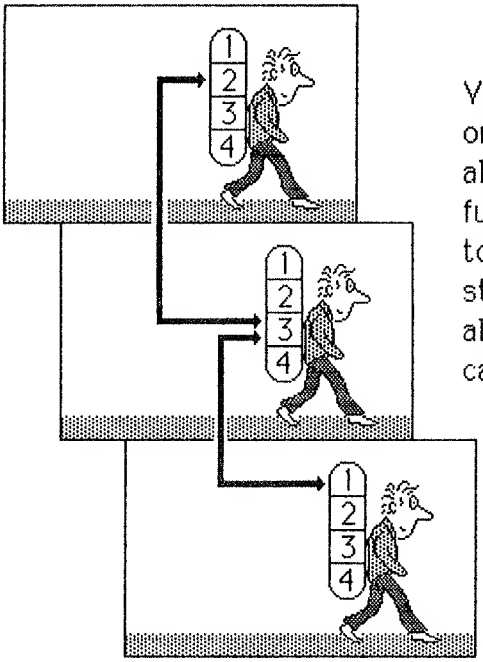


Figure 1

Another concept which is easier to understand with this same analogy is parameters being passed to several levels of procedures/functions. At each level, the parameter may be refered to as a different parameter number, depending on where it was encountered in the parameter list. All changes, at all levels, are retained and returned to the main program. Figure two shows how the same parameter may be refered to with different numbers at each level.



Values passed to a procedure or function may be passed along to other procedures or functions. Any changes made to these parameters, at any stage, will be returned back all the way to the original caller.

Figure 2

These examples show just how versatile procedures and functions are. It is important that the user understand these two statements fully to obtain the most from the NADOL language.

The examples shown have dealt with the passed parameters being integers. This is the assumed condition for a passed parameter unless otherwise specified. If the **%n** is followed by a '**B**', the parameter will be referenced as though it was declared as a byte variable. In either case, a subscript may be added to the variable if the passed parameter from the main program is an array variable. The subscript specifies the element within the array to use. If no subscript is used with an array variable, the first element will be used.

No type checking is done on this mechanism, so care should be taken to make sure that passed variables match the way that they are used in a procedure or function, unless some other effect is desired.

Procedures and functions are a much more powerful type of subroutine mechanism than BASIC's GOSUB system. Since parameters may be passed, subroutines may operate on a variety of variables and parameters without having to be dedicated to just one specific variable or parameter.

For those familiar with Pascal or C, this parameter system will seem familiar, since those languages operate in much the same fashion.

Chapter 7

Example Programs

This chapter describes each of the example programs supplied with NADOL in detail. Following the flow of each program is an excellent way for the user to become more familiar with NADOL. We recomend that you experiment with the programs supplied. Try various changes and view their effects. It is a good idea to first make a copy of your master disk before saving any changes that you may make. See the prologue to the NADOL manual for instructions on duplicating your original diskette.

In order to demonstrate the flexibility of NADOL, the first example will have nothing at all to do with diskettes. It is an entertainment program, of sorts, which will play the familiar game of brick-out. As simple a task as this may sound, most of the flow control statements, and a great number of the built in procedures are used at one point or another in this program, making it an ideal example.

The comments to the right of the listing are no present in the actual program on your diskette. They are provided as a running commentary of the operations being performed at each step in the program.

```
; LORES BRICK-OUT GAME

DEFINE INTEGER CURRENTX,CURRENTY          Define all of the variables
DEFINE INTEGER SIDECOL,PDLCOL,BALLCOL     which will be used during
DEFINE INTEGER DELTAX,DELTAY,PADDLE       the program.
DEFINE INTEGER OLDPDL,OLDX,OLDY,SCANVAL   It is a good idea to use
DEFINE INTEGER HIT,OLDSCORE,SCORE,GAME    longer names which make
DEFINE INTEGER OURY,OURX,KEY              sense to the user.

SIDECOL=4                                 Set the color of the border.
PDLCOL=5                                  The color of the 'paddle'.
BALLCOL=13                                The color of the 'ball'.


                                          This procedure draws a
PROCEDURE DRAWLINE                        line of colored dots.
  OURY=1                                  Start at row 1.
  WHILE OURY<39                           Continue to the bottom.
    COLOR=(OURY/2 MOD 2)+1                Set the color of the dot.
    VLINE(#1,OURY,OURY+2)                 Draw a two block line.
    OURY=OURY+2                           Move down 2 for the next
  ENDWHILE                                block, and loop until done.
ENDPROC                                   End this procedure.
```

```
PROCEDURE DRAWSCAN                  This procedure draws the
                                    game board, using the
  LORES                             DRAWLINE procedure.
  OURX=20                           Set LORES graphics mode.
  WHILE OURX<35                     Start at column 20.
    DRAWLINE(OURX)                  Draw until column 35.
    OURX=OURX+2                     Draw the current line.
  ENDWHILE                          Step by 2.
  COLOR=SIDECOL                     Loop back until all done.
  HLINE(0,0,39)                     Set the new color.
  VLINE(39,0,39)
  HLINE(0,39,39)                    Draw the box around the
  VLINE(0,0,39)                     playing field.
ENDPROC
                                    End this procedure.

                                    Now that all of the
                                    variables and procedures
                                    have been defined, we
GAME=1                              begin the main program.
WHILE 1                             Causes a loop forever.
  IF GAME                           If a new game is starting...
    HOME                            Clear the screen.
    DRAWSCAN                        Draw the playing field.
    BEEP(200,40)                    Notify the user via a beep.
    CURRENTX=15
    CURRENTY=11                     Set the starting conditions.
    DELTAX=-1
    DELTAY=0
    OLDPDL=1
    HIT=1
    OLDX=4
    OLDY=4
    SCORE=0
    OLDSCORE=1
    GAME=0
  ENDIF
```

```
IF SCORE<>OLDSCORE                          If the score has changed...
  GOTOXY(4,21)
  PRINT "SCORE=";SCORE                      Print the new score.
  OLDSCORE=SCORE
ENDIF

PADDLE=PDL(0)/7+1                           Read the game controller.

IF PADDLE>33                                If the value is too large
  PADDLE=33                                 for our needs, chop it back
ENDIF                                       down to size.

                                            If the paddle has moved,
IF (PADDLE<>OLDPDL) OR (HIT<>0)             the ball hit the paddle...
  COLOR=0
  VLINE(1,OLDPDL,OLDPDL+5)                  Erase the old paddle.
  OLDPDL=PADDLE                             Remember the new value.
  COLOR=PDLCOL
  VLINE(1,PADDLE,PADDLE+5)                  Draw the new paddle.
  HIT=0                                     Clear the 'hit' flag.
ENDIF

CURRENTX=CURRENTX+DELTAX                    Move the ball in the X and
CURRENTY=CURRENTY+DELTAY                    Y directions.

IF CURRENTY<1                               If Y is too small...
  CURRENTY=1                                Put it back on the screen.
ENDIF
IF CURRENTY>38                              If Y is too large...
  CURRENTY=38                               Put it back on the screen.
ENDIF
IF CURRENTX<1                               If X is too small...
  CURRENTX=1                                Put it back on the screen.
ENDIF
IF CURRENTX>38                              If X is too large...
  CURRENTX=38                               Put it back on the screen.
ENDIF
```

```
SCANVAL=LSCAN(CURRENTX,CURRENTY)        Read color under the ball.
COLOR=0
PLOT(OLDX,OLDY)                         Erase the old ball position.
COLOR=BALLCOL
PLOT(CURRENTX,CURRENTY)                 Plot the new ball position.

IF OLDX=1                               If ball was in column 1...
  HIT=1                                 Then we may have hit the
ENDIF                                   paddle.

OLDX=CURRENTX                           Remember the new
OLDY=CURRENTY                           ball position.

IF CURRENTX=1                           If ball is in column 1...
  IF (CURRENTY<PADDLE) OR (CURRENTY>PADDLE+4) OR (SCORE=152)
    BEEP(100,50)                        Game may be over.
    GOTOXY(20,21)
    IF SCORE<5                          Print out the final score
      PRINT "UNCOORDINATED"             and give the user a
    ELSE                                rating of his/her
      IF SCORE<20                       performance based on
        PRINT "TRY HARDER!"             that score.
      ELSE
        IF SCORE<50
          PRINT "NOT BAD..."
        ELSE
          IF SCORE<100
            PRINT "GETTING BETTER"
          ELSE
            IF SCORE<152
              PRINT "ABOVE AVERAGE"
            ELSE
              PRINT "EXCELLENT!"
            ENDIF
          ENDIF
        ENDIF
      ENDIF
    ENDIF
```

```
      GOTOXY(2,23)                          Wait for a key to be pressed.
      PRINT "PRESS ANY KEY TO PLAY AGAIN";
      KEY=READ(1)
      GAME=1                                Then start over again.
    ELSE                                    The ball was hit correctly.
      BEEP(170,2)                           Make a noise.
      DELTAX=-DELTAX                        Reverse the X direction.
      DELTAY=CURRENTY-PADDLE-3              Set the new Y speed.
      IF DELTAY>=0
        DELTAY=DELTAY+1
      ENDIF
    ENDIF
  ENDIF

  IF (SCANVAL=1) OR (SCANVAL=2)             If we hit a block...
    COLOR=0                                 Erase all of it.
    VLINE(CURRENTX,(CURRENTY-1)/2*2+1,(CURRENTY-1)/2*2+2)
    SCORE=SCORE+1                           Give the user a point.
    BEEP(250,2)                             Make a noise.
    DELTAX=-DELTAX                          Reverse direction.
  ELSE                                      Otherwise, if ball is moving
    IF (CURRENTX=38) AND (DELTAX=1)         forward and at back...
      BEEP(70,2)                            Make a noise.
      DELTAX=-DELTAX                        Head back to the user.
    ENDIF
  ENDIF

  IF (CURRENTY=1) OR (CURRENTY=38)          If at upper or lower edge...
    BEEP(70,2)                              Make a noise.
    DELTAY=-DELTAY                          Go the other direction.
  ENDIF

ENDWHILE                                    Always loop back for more.
```

Some possible ideas for instructional changes that the user could make would be:

1) Multiple balls per game.
2) Two player game (Think about using two element arrays).
3) Multiple skill levels.

The possibilities are almost limitless!

Our second example is a color disk mapper. It will display the Volume Table Of Contents (VTOC) for a DOS 3.3 diskette in color on the screen. Then it proceeds to display the location of all of the sectors used by each file on the disk in different colors. This program is useful and educational because it shows how easily NADOL can be applied to a real world application.

```
DEFINE INTEGER TRACK,SECTOR,COUNT          Define all variables.
DEFINE INTEGER ERR,Y,TEMP,LOC,POS,COL
DEFINE INTEGER DTRACK,DSECTOR,TTRACK,TSECTOR
DEFINE BYTE[2] COLORS
DEFINE BYTE[40] NAME
DEFINE BYTE[255] VBUFF,TBUFF,CBUFF

LORES                                      Set LORES graphics.
COLORS[1]= 10                              Set the two colors used
COLORS[2]= 15                              to display the VTOC.

SETFORMAT(0,ADDR16,DATA16,INT16)           Use DOS 3.3 format/interlv.

TRACK=$11                                  VTOC is on track $11,
SECTOR=$0                                  sector 0.
COUNT=1                                    We're reading one sector.
RSECT(VBUFF,TRACK,0,SECTOR,COUNT,6,1,ERR)  Read in the sector
TRACK=0                                    Now start at track 0.
WHILE TRACK<=34                            Loop until track 34.
  TEMP=VBUFF[$39+TRACK*4]+VBUFF[$38+TRACK*4]*256   Get track bits.
```

```
  IF (TEMP=0) OR (TEMP=$FFFF)          Is it all used or unused?
    IF TEMP=0                          If all used...
      COLOR=COLORS[1]                  Use color 1.
    ELSE                               Otherwise...
      COLOR=COLORS[2]                  Use color 2
    ENDIF
    HLINE(0,TRACK,15)                  Draw a line for whole track.
  ELSE                                 We have to plot each one...
    Y=0
    WHILE Y<16                         Loop for all sectors.
      IF TEMP MOD 2                    If the sector is free...
        COLOR=COLORS[2]                Use color 2.
      ELSE                             Otherwise...
        COLOR=COLORS[1]                Use color 1.
      ENDIF
      TEMP=TEMP/2                      Get rid of that bit.
      PLOT(Y,TRACK)                    Plot the dot.
      Y=Y+1                            Move to the next dot.
    ENDWHILE
  ENDIF
  TRACK=TRACK+1                        Move to the next track.
ENDWHILE
                                       Now the VTOC is plotted,
                                       so we move on to plotting
                                       the file locations.
COL=1                                  Start with color 1.
COLOR=COL                              Make it the current color.
DTRACK=$11                             Directory starts on track $11
DSECTOR=$F                             sector $F.
WHILE (DTRACK<>0) AND (DSECTOR<>0)     Run until the links are 0...
  COUNT=1
  RSECT(CBUFF,DTRACK,0,DSECTOR,COUNT,6,1,ERR)    Read the sector.
  POS=11                                         Start at position 11.
  WHILE (CBUFF[POS]<>0) AND (POS<255)  Until all names used up...
    IF CBUFF[POS]<>$FF                 If not a deleted file...
      TTRACK=CBUFF[POS]                Get track for TSL.
      TSECTOR=CBUFF[POS+1]             Get sector for TSL.
      COPY(CBUFF[POS+3],NAME,25)       Move the name.
      PRINT !NAME                      Print the name.
      COUNT=1
      RSECT(TBUFF,TTRACK,0,TSECTOR,COUNT,6,1,ERR)    Read the sector.
```

```
      LOC=12                                  Start at position 12.
      PLOT(TSECTOR+20,TTRACK)                 Mark the TSL sector.
      WHILE (LOC<255) AND ((TBUFF[LOC]<>0) OR (TBUFF[LOC+1]<>0))
        PLOT(TBUFF[LOC+1]+20,TBUFF[LOC])      Plot each sector of the file.
        LOC=LOC+2
      ENDWHILE
      COL=(COL+1) MOD 15                      Move to the next color.
      IF COL=0                                Skip color 0, since black
        COL=1                                 doesn't show up.
      ENDIF
      COLOR=COL                               Set new color.
    ENDIF
    POS=POS+$23                               Move to next name.
  ENDWHILE
  DTRACK=CBUFF[1]                             Locate next track/sector
  DSECTOR=CBUFF[2]                            for rest of catalog.
ENDWHILE
```

This program shows how a very small program can perform complex tasks involving the disk drivce with very little effort. This program shows how a directory can be followed to look at the information for each and every file on the disk. In this case the information was displayed graphically on the screen. But it could just as easily have been used to edit each program and the way that each was arranged on the diskette.

Some enhancements which could be added to this program are:
1) Ask the user to insert a disk before starting.
2) Use HIRES graphics for the display.
3) Automatically mark tracks $0-$2 and $11 as used.

Another source of programs to examine are the programs which make up Nibbles Away III and its associated utilities. Each of these may be loaded into the editor and viewed just like any other program.

Chapter 8

Built-in Statements

The following chapter describes all of the built-in functions, procedures and statements in NADOL. Many examples are provided to make the function of each clearer. In order to avoid confusion, the following layout is used for each description:

NAME TYPE

Purpose: A short description of the statement's function.

Syntax: The syntax for using the statement goes here. Braces { } are used to identify optional fields.

Remarks: A description of any parameters, and any additional helpful information about the statement.

Any parameters enclosed in double triangles «» MUST be names of variables when the statement is used.

Example: This section provides a view of the actual usage of the statement and what its effect is.

Appendix XX provides a quick-reference to all of the statements described in the chapter.

AUXMOVE PROCEDURE

Purpose: Move data to or from the auxiliary memory in an Apple //e or and Apple //c.

Syntax: **AUXMOVE(apple•addr,aux•addr,length,direction)**

Remarks:

«apple•addr»	The name of the variable which specifies the starting address in the apple for the move.
aux•addr	A number in the range $0-$B7FF. This is the starting address in auxiliary memory for the move.
length	The number of *bytes* to move to or from auxiliary memory.
direction	Specifies whether data should be transferred to or from auxiliary memory: 0= From auxiliary memory 1= To auxiliary memory

This procedure can make use of up to 46k of the available auxiliary memory.

BEEP PROCEDURE

Purpose: Sounds a tone from the built-in speaker.

Syntax: **BEEP(tone,time)**

Remarks:

tone	This is a number from 0 to 255 which determines the frequency of the sound. Zero is the lowest frequency, 255 is the highest.
time	This is a number from 0 to 255 which determines the duration of the tone. Zero is the shortest, 255 is the longest.

Example: The following program:

```
DEFINE INTEGER I
I= 50
WHILE I<=100
   BEEP(I,30)
   I=I+10
ENDWHILE
```

Will cause 6 tones to be produced, each one higher in pitch than the last.

CALL PROCEDURE

Purpose: Executes a machine language subroutine.

Syntax: CALL(address,accumulator,x•register,y•register,status)

Remarks:

«address» The name of the variable which specifies the address of the subroutine to be executed.

«accumulator» The name of the variable containing the value to be passed in the accumulator. On exit the accumulator value will be returned here.

«x•register» The name of the variable containing the value to be passed to the X register.

«y•register» The name of the variable containing the value to be passed to the Y register.

«status» The name of the variable containing the value to be passed to the status register.

The routine being called should end with an RTS instruction. Zero page locations $0 through $1F are available for use by the routine. Any other memory should be used with care.

Example: The following program:

```
DEFINE INTEGER ACC,X,Y,STAT
X=1
CALL(MEMORY[$FB1E],ACC,X,Y,STAT)
PRINT Y
```

Would call the monitor paddle read routine at $FB1E. This routine returns the paddle value in the Y register which would be placed in the variable Y. Printing Y prints the paddle value.

Note that even though no values are passed to the accumulator or the status register, variables must still be used for these parameters.

CATALOG STATEMENT

Purpose: Displays a list of all files on a data diskette, and the amount of free space on that diskette.

Syntax: **CATALOG**

Remarks: The **WORKDRIVE** command can be used to select the slot and drive which will be used by this command.

CLEAR STATEMENT

Purpose: Clears all variables.

Syntax: **CLEAR**

Remarks: This statement also clears all user defined procedures and functions, so if it is used from within a program, all user defined procedures and functions will become inaccessible.

CLREOL STATEMENT

Purpose: Clears all text to the right of the current cursor location.

Syntax: **CLREOL**

Remarks: This command does not move the cursor.

CLREOP STATEMENT

Purpose: Clears all text to the right and below the current cursor location.

Syntax: CLREOP

Remarks: This command does not move the cursor.

COLOR= STATEMENT

Purpose: Sets the color used for low resolution graphics.

Syntax: **COLOR= expression**

Remarks: The value of expression may be from 0 to 15, corresponding to the following colors:

0= Black	8= Brown
1= Magenta	9= Orange
2= Dark Blue	10= Gray 2
3= Purple	11= Pink
4= Dark Green	12= Light Green
5= Gray 1	13= Yellow
6= Medium Blue	14= Aquamarine
7= Light Blue	15= White

If expression is greater than 15 then expression mod 16 is used for the color value.

Example: The following program:

```
DEFINE INTEGER DIR,POS,COL
DIR=1
LORES
WHILE 1
   COLOR=COL
   HLINE(0,POS,39)
   VLINE(POS,0,39)
   COL=(COL+1) MOD 16
   POS=POS+DIR
   IF (POS=0) OR (POS=39)
      DIR=-DIR
   ENDIF
ENDWHILE
```

Would draw a series of horizontal and vertical bars back and forth on the screen until ctrl-c is pressed to halt the program.

Note the use of the **WHILE 1** statement to create a loop which will never terminate, giving a continuously running program.

CONVERT PROCEDURE

Purpose: Converts a byte array containing ASCII text into an array of hexadecimal or decimal values.

Syntax: **CONVERT(source,destination,type,size,count1,count2)**

Remarks:

«source» The name of the byte array containing the ASCII text to be scanned. The text must be terminated by a zero value.

«destination» The name of the array where the converted data will be placed.

type Determines whether scanning should take place in decimal or hexadecimal:
0= Hexadecimal 1= Decimal

size Determines if the values placed in **dest** should be bytes or integers:
0= Integers 1= Bytes

«count1» The name of the variable where the number of characters scanned **source** will be returned.

«count2» The name of the variable where the number of values placed into **dest** will be returned.

This procedure can be used to convert a string entered by a user into a data array which is more readily usable by the other built-in functions.

Example: The following program:

```
DEFINE BYTE[30] STRING,ARRAY
DEFINE INTEGER C1,C2
PACK STRING WITH "D5 AA 96"
CONVERT(STRING,ARRAY,0,1,C1,C2)
DISPLAY(ARRAY,C2)
```

Would print:

```
D5 AA 96
```

Showing that the text string had been converted to hexadecimal values in ARRAY which are usable by NADOL.

COPY PROCEDURE

Purpose: Copies a block of data from one location to another.

Syntax: **COPY(source,destination,length)**

Remarks:

«source» The name of the variable from which the data should be copied.

«destination» The name of the variable where the copied data will be placed.

length The number of *bytes* to be copied.

This procedure can copy data between any two locations in the machine's memory, whether they are within the same variable or not.

Example: The following program:

```
DEFINE BYTE[30] MINE,YOURS
PACK MINE WITH "THIS IS MY STRING"
COPY(MINE[11],YOURS,6)
PRINT !YOURS
```

Would print:

```
STRING
```

Showing that the six bytes (5 characters plus the terminating zero for a string) starting at position 11 in MINE were copied to YOURS at its start.

DEFINE STATEMENT

Purpose: Allocates space for one or more variables.

Syntax: **(DEFINE) type ([n]) name (,name).....**

Remarks:

type The type of variable being defined, either **BYTE** or **INTEGER**.

«name» The name of the variable. It can be up to 8 characters in length.

The optional subscript (**[n]**) defines the number of elements to be set aside for this variable. The number specified will be the highest element set aside, including element zero, giving **n+1** elements total. If the subscript is left off, 1 element is set aside, and array operations may not be performed on the variable.

A variable of the specified type will be created for each of the names listed, and then all of its elements are set to zero.

When running a program, NADOL scans the entire program first to find all of the **DEFINE** statements. This means that **DEFINE** statements may be anywhere in a program, before or after they are referenced.

Example: The following statement:

```
DEFINE INTEGER[8] MINE,HIS,HERS
```

Would create three 9 element (0-8) integer arrays, one named **MINE**, one named **HIS** and one named **HERS**.

The statement:

```
BYTE SINGLE
```

Would create a variable named **SINGLE** which would consists of one element of one byte.

DELAY STATEMENT

Purpose: Pauses for a specific amount of time.

Syntax: **DELAY(expression)**

Remarks: **expression** The number of milliseconds to pause.

Example: The statement:

```
DELAY(40*500)
```

Would delay 20000 milliseconds, which is 20 seconds.

DELETE COMMAND

Purpose: Removes a file from the current work disk.

Syntax: **DELETE filename**

Remarks: **filename** Either a name of up to twelve characters enclosed in quotes or the name of a variable which is a byte array containing a file name.

This procedure will remove a file and reclaim the space used by it on the current diskette. Care should be taken with this command since once a file is deleted, it cannot be recovered.

DISASM STATEMENT

Purpose: Displays a disassembled listing of machine code.

Syntax: **DISASM(start,label,lines,offset)**

Remarks:

«start» The name of the variable which defines the first location to be disassembled.

label The value which will be shown at the left edge of the disassembly and used for the display of relative addresses.

lines The number of lines to display.

«offset» The name of the variable where the offset of the next instruction from the start of the current disassembly will be placed. This is used to determine the next location to disassemble.

On an Apple //c all of the extra instructions of the 65C02 microprocessor will be correctly displayed.

Example: The following statements:

```
DEFINE INTEGER LOC,OFFSET
LOC=$F800
DISASM(MEMORY[LOC],LOC,10,OFFSET)
```

Would display the following:

```
F800-  4A          LSR
F801-  08          PHP
F802-  20 47 F8    JSR   $F847
F805-  28          PLP
F806-  A9 0F       LDA   #$0F
F808-  90 02       BCC   $F80C
F80A-  69 E0       ADC   #$E0
F80C-  85 2E       STA   $2E
F80E-  B1 26       LDA   ($26),Y
F810-  45 30       EOR   $30
```

Then the statements:

```
LOC=LOC+OFFSET
DISASM(MEMORY[LOC],LOC,10,OFFSET)
```

Would display the next 10 instructions, which would be:

```
F812-  25 2E       AND   $2E
F814-  51 26       EOR   ($26),Y
F816-  91 26       STA   ($26),Y
F818-  60          RTS
F819-  20 00 F8    JSR   $F800
F81C-  C4 2C       CPY   $2C
F81E-  B0 11       BCS   $F831
F820-  C8          INY
F821-  20 0E F8    JSR   $F80E
F824-  90 F6       BCC   $F81C
```

This shows how **offset** can be used to display multiple sections of sequential code.

DISPLAY PROCEDURE

Purpose: Displays a block of data in hexadecimal.

Syntax: **DISPLAY(start,length)**

Remarks: **«start»** The name of the variable which defines the first location to be displayed.

length The number of *bytes* to display.

The bytes will be displayed with a 4 digit label, in groups of 16 bytes per line. Pressing the **SPACE** bar will pause the listing, and ctrl-C will stop the listing.

Example: The statement:

```
DISPLAY(RBUF,256)
```

Would show the first 256 bytes of **RBUF** on the screen.

EDIT COMMAND

Purpose: Invokes the built-in program editor.

Syntax: **EDIT**

Remarks: See chapter 4 for full instructions.

FILL PROCEDURE

Purpose: Fills a section of memory with a value.

Syntax: **FILL(start,length,value)**

Remarks:

«start» The name of the variable which determines the start of the area to be filled.

length The number of *bytes* to fill.

value The value from 0 to 255 to fill with.

Example: The statement:

```
FILL(RBUF[200],100,$05)
```

Would place the value **$05** in 100 bytes of **RBUF** starting with the 200th element of **RBUF**.

FIND PROCEDURE

Purpose: Finds a specified pattern with the ability to ignore bit 7 and perform wildcard matching.

Syntax: **FIND(start,length1,pattern,length2,7•flag,wild•flag,offset)**

Remarks:

«start» The name of the variable which determines the start of the area to search through.

length1 The number of *bytes* to search.

«pattern» The name of the variable containing the pattern to search for.

length2 The length of the pattern, in *bytes*

7•flag Determines whether bit 7(the most significant bit) will be used in the comparison.
0= Bit 7 is used 1= Bit 7 is ignored

wild•flag Determines whether wild cards will be used during the search. A 'zero' value disables wildcards. A 'one' value enables wild cards, causing any zero values in **pattern** to match any value in the range being checked.

«offset» The name of the variable where the offset of the matched pattern will be returned. If no match is found, -1 will be returned.

FINDs using either 'bit 7 ignore' or 'wild cards' take longer to execute than those without.

Example: The following program:

```
BYTE[50] YOUR
BYTE[8] SPOT
INTEGER POS

PACK YOUR WITH "HERE WE GO AGAIN TO THE STORE"
PACK SPOT WITH "AGAIN"
FIND(YOUR,LENGTH(YOUR),SPOT,LENGTH(SPOT),0,0,POS)
PRINT POS
```

Would print the number 11, since the word 'again' was found at the eleventh position in **YOUR**.

FLASH STATEMENT

Purpose: Sets flash mode for printed characters.

Syntax: **FLASH**

Example: The following program segment:

```
NORMAL
PRINT "HELLO";
FLASH
PRINT "THERE";
INVERSE
PRINT "HOW ARE ";
NORMAL
PRINT "YOU?"
```

Would print **HELLO** normally, **THERE** flashing, **HOW ARE** in inverse and **YOU?** normally.

FILTER PROCEDURE

Purpose: Copies data into the write buffer, passing it through a 'filter' to remove unwanted values.

Syntax: **FILTER(«start»,length,«table»,«number»)**

Remarks:

«start» The name of the variable which defines the start of the location to begin taking data from.

length The number of bytes to pass through the 'filter'.

«table» The name of the variable specifying the 'filter' table to use.

«number» The number of bytes filled in the write buffer.

This procedure is used to move data from the read buffer to the write buffer prior to sending it to the disk drive. The 'filter' allows all bytes which are not valid disk bytes to be removed from the data.

A 'filter' is a table in memory, usually created with the **MAKE** procedure. It is 256 bytes in length, one location for each possible value of a byte. If the contents of a location is 0, then that byte will not be passed to the write buffer. If the contents is non-zero, then the contents of the table will be passed to the write buffer. This actually allows bytes to be translated into other bytes as they are written out to the diskette.

FORMAT PROCEDURE

Purpose: Formats a range of tracks.

Syntax: **FORMAT(first,last,volume,interleave,nsect,slot,drive,error)**

Remarks:

first The first track to format.

last The last track to format.

volume The volume number to be recorded on the diskette.

«interleave» The name of the byte array containing the numbers for the sectors on the tracks to be formatted.

nsect The number of sectors per track.

slot The slot of the disk drive to format.

drive The drive number to format.

«error» The name of the variable to return the result code in. A value of 0 signifies that no errors took place.

Note that this command will erase any data previously present on the track(s) being formatted.

Example: The statement:

```
FORMAT(0,34,0,FORM16,16,6,1,ERR)
```

Would format the entire disk in slot six, drive one, using the predefined variable **FORM16** for the interleave table, with 16 sectors per track and return any error code in the variable **ERROR**.

FREE FUNCTION

Purpose: Returns the amount of space available for programs and data.

Syntax: **variable= FREE(x)**

Remarks: **x** is a dummy expression used only as a place holder. 0 is the normal value used.

Example: The statement:

```
PRINT FREE(0)
```

Would print the amount of available space for additional data and program.

FUNCTION STATEMENT

Purpose: Defines a user subroutine which returns a value to the calling program.

Syntax: **FUNCTION name**

Remarks: Chapter 6 contains full details on the use of procedures and functions.

GOTO STATEMENT

Purpose: Transfers program execution to another location.

Syntax: **GOTO labelname**

Remark: Program flows begin following the specified label statement.

The GOTO statement is provided only for those cases where it is absolutely necessary. The structured nature of NADOL allows almost all programming situations to be overcome without using the GOTO statement.

Example: The following program:

```
LABEL THERE
PRINT "FOR EVER AND EVER"
GOTO THERE
```

Would print **FOR EVER AND EVER** continuously on the screen until the program was stopped with a ctrl-C.

GOTOXY PROCEDURE

Purpose: Moves the cursor to a new location on the screen.

Syntax: GOTOXY(x,y)

Remarks: x The new horizontal position in the range 0 to 39.

y The new vertical position in the range 0 to 23.

This procedure does not place a cursor on the screen, it just sets the next location for an input or print operation.

Example: The following program:

```
DEFINE INTEGER Y
Y=0
WHILE Y<10
   GOTOXY(20,Y)
   PRINT Y
   Y=Y+1
ENDWHILE
```

Would print the numbers 0 through 9 on consecutive lines of the screen, indented twenty spaces.

HCOLOR PROCEDURE

Purpose: Sets the color for high resolution plotting.

Syntax: **HCOLOR= expression**

Remarks: **expression** is a value from 0 to 7 corresponding to the following colors:

0= Black1	4=Black2
1= Green	5=Orange
2= Violet	6=Blue
3= White1	7=White2

If **expression** is greater than 7, **expression** MOD 8 is used for the color value.

Example: The following program:

```
DEFINE INTEGER X
HIRES
X=0
WHILE X<280
  HCOLOR= X/8
  HPLOT X,0 TO X,191
  X=X+1
ENDWHILE
```

Would draw a series of vertical bars on the hires screen, each one 8 dots wide.

HEXPACK STATEMENT

Purpose: Reads HEX data into a byte array, with an optional checksum.

Syntax: **HEXPACK name WITH "text" (, checksum)**

Remarks:

«name» The name of the array where the converted data will be placed.

text A series of ASCII characters which represent the hexadecimal form of the data which is to be placed in **name**.

checksum An optional value which is used to verify that the characters in **text** have been typed in correctly.

This statement is normally used to enter raw data, or machine language routines. Since the main use of this statement will be for entering large amounts of raw data, the checksum has been provided as an optional method of verifying that no typographical errors exist. The checksum is computed by stepping through the each value, exclusive ORing each with the previous, and adding one to the result at each stage.

Example: The following program:

```
DEFINE BYTE[40] DATA
HEXPACK DATA WITH "C8C5CCCCCF00",$C2
PRINT !DATA
```

Would print:

```
HELLO
```

On the screen, since that was the ASCII equivalent of the HEX data which was entered.

HIRES STATEMENT

Purpose: Initializes high resolution graphics mode.

Syntax: **HIRES**

Remarks: This command turns on the hires screen and clears it to black. This command should be used before any hires graphic commands are performed, or unpredictable results may occur.

Applesoft basic in ROM is required to use this statement.

Example: See the **HCOLOR** statement.

HLINE PROCEDURE

Purpose: Draws a line on the lores screen.

Syntax: **HLINE(x1,y1,x2)**

Remarks: **x1** X coordinate of the starting point, in the range 0-39.

y1 Y coordinate of the starting point, in the range 0-47.

x2 X coordinate of the ending point, in the range 0-39.

x2 must be greater than **x1**.

Example: See the **COLOR** statement.

HOME STATEMENT

Purpose: Clears the screen and places the cursor in the upper left hand corner of the screen.

Syntax: **HOME**

HPLOT STATEMENT

Purpose: Plots points or draws lines on the hires screen.

Syntax: **HPLOT (x,y) (TO x,y)....**

Remarks: **x** The x coordinate in the range 0-279.

y The y coordinate in the range 0-191.

If only one coordinate is specified then a dot is plotted.

If only a **TO x,y** is specified then a line will be drawn from the last plotted point.

If a coordinate and one or more secondary coordinates is specified then a string of connected lines will be drawn between all of the specified points.

Example: See the **HCOLOR** statement.

HSCRN FUNCTION

Purpose: Returns the value of a dot on the hires screen.

Syntax: **variable= SCRN(x,y)**

Remarks: **x** The x coordinate of the point to read, in the range of 0 to 279.

y The y coordinate of the point to read, in the range of 0 to 191.

The value returned will be a 1 or a 0, depending on whether the specified dot was on or off. The color of the dot is not returned, since colors are determined by whether or not dots are next to each other, and whether they are in odd or even columns.

IF, ELSE and ENDIF STATEMENT

Purpose: Alters program flow based on a condition.

Syntax:

```
IF expression
   .
   ( statements executed on true )
   .
( ELSE )
   .
   ( statements executed on false )
   .
ENDIF
```

Remarks: If the value of **expression** is non-zero, then the statements up to the **ENDIF** or up to the optional **ELSE** will be executed. If **expression** is zero, then the statements following the **ELSE** statement will be executed if it is present, otherwise program execution will continue following the **ENDIF**.

IF-ELSE-ENDIF blocks may be nested up to 8 levels deep.

Example: The following program segment:

```
IF YOU > 10
   PRINT "GREATER THAN 10"
ELSE
   PRINT "LESS THAN OR EQUAL TO 10"
ENDIF
```

Would print **GREATER THAN 10** if the variable **YOU** was greater than 10, or **LESS THAN OR EQUAL TO 10** otherwise.

IN# STATEMENT

Purpose: Takes program input from a peripheral slot.

Syntax: **IN# expression**

Remarks: **expression** may be a value from 0 to 7, and corresponds to the peripheral slots in the computer. A value of 0 will return input to the keyboard.

Example: The statement:

```
IN# 2
```

Would instruct NADOL to begin taking input from the peripheral in slot number 2.

INIT COMMAND

Purpose: Formats a diskette for storage of programs and data.

Syntax: **INIT name**

Remarks: The **name** specified may be up to 12 characters enclosed in quotes or the name of a byte array containing a file name. It will be placed on the diskette and displayed whenever a **CATALOG** command is performed.

The current work drive is used for this command.

Example: The statement:

```
INIT "NEWDISK"
```

Would format a diskette in the current workdrive, making it available for storage of NADOL data. The name on the disk would be set to **NEWDISK**.

INPUT PROCEDURE

Purpose: Reads ASCII data from the keyboard into a byte array.

Syntax: **INPUT(name,max,count)**

Remarks:

«name» The name of the byte array to place the data read from the keyboard into. The text will be terminated by a zero.

max The maximum number of characters to allow.

«count» The name of the variable where the count of the number of characters read is to be returned.

During an input statement, the user may use the backspace key to delete errors.

The **<RETURN>** key accepts a response.

If the user types an **<ESC>**, the input will abort and set **count** to -1.

If the user attempts to type in more than **max** characters, the extras will be ignored and the cursor will not move on the screen.

Example: The statement:

```
INPUT(EMPTY,20,COUNT)
```

Would prompt the user for a string with a limit of 20 characters. The data typed by the user would be placed into the array named **EMPTY**, and the number of characters entered would be returned in **COUNT**.

INVERSE STATEMENT

Purpose: Sets inverse mode for all printed characters.

Syntax: **INVERSE**

Example: See **FLASH**.

LABEL STATEMENT

Purpose: Sets a location which can be branched to with a **GOTO** statement.

Syntax: **LABEL name**

Remarks: **name** is up to eight characters which define the name of the location marker being set.

A label may not be defined within an **IF-ELSE-ENDIF** or **WHILE-ENDWHILE** block.

Example: See the **GOTO** statement.

LCMOVE PROCEDURE

Purpose: Moves data to or from a language card.

Syntax: **LCMOVE(mem•address,lc•address,length,direction)**

Remarks:

«mem•address»	The name of the variable which specifies the start of the area to transfer.
lc•address	The value of the starting location for the transfer, in the range $0-$2FFF.
length	The number of *bytes* to transfer to or from the language card.
direction	Specifies whether data is transferred to or from the language card: 0= From language card 1= To language card

LENGTH FUNCTION

Purpose: Returns the length of the text in a byte array.

Syntax: **variable= LENGTH(name)**

Remarks:

«name»	The name of a byte array containing ASCII text terminated with a zero.

Example: The following program:

```
DEFINE BYTE[30]:LUNCH

PACK LUNCH WITH "SANDWICHES AND SODA"
PRINT LENGTH(LUNCH)
```

Would print the value **19**, which is the length of the string placed into the variable **LUNCH**.

LIST COMMAND

Purpose: Displays the current program.

Syntax: LIST

Remarks: The **<SPACE>** bar will pause the listing and ctrl-C will abort the listing.

LOAD COMMAND

Purpose: Loads a file from the current work diskette.

Syntax: **LOAD filename (AT address)**

Remarks:

filename Up to twelve characters in double quotes, or the name of the byte array containing the name of the file to load.

«address» The name of the variable defining the location at which to begin loading the data.

If the AT option is left off, the file is loaded as a program into the current program space and all variables are cleared.

The AT option specifies an address at which to load the specified file. In this case, the variables are not cleared. This is useful for loading data such as hires pictures.

Example: The following statement:

```
LOAD "PROGRAM"
```

Would load the file named **PROGRAM** from the current workdrive.

The statement:

```
LOAD "PICTURE" AT MEMORY[$4000]
```

Would load the file named **PICTURE** at memory address $4000.

If a **LOAD** statement which loads a program is executed from within a running program, the file will be loaded normally, all variables will be cleared, and the new program will begin running at its start.

This allows one program to transfer control to another.

LORES STATEMENT

Purpose: Initializes lores graphics.

Syntax: **LORES**

Remarks: Enables and clears the lores screen. This command should be executed before performing any lores graphics commands.

The cursor will be moved to the lower left hand corner of the screen when this command is executed.

Example: See the **COLOR** statement.

LSCRN FUNCTION

Purpose: Returns the color of a point on the screen.

Syntax: **variable= LSCRN(x,y)**

Remarks: **x** The x coordinate of the point to check, in the range 0 to 39.

y The y coordinate of the point to check, in the range 0 to 47.

This function returns a value from 0 to 15, corresponding to the colors listed in the table under the **COLOR** statement.

MAKE PROCEDURE

Purpose: Creates a filter in the specified array.

Syntax: **MAKE(address,length,start,num•zeroes,bit•length)**

Remarks:

«address»	The name of the variable which determines the start of the area in which to place the filter.
length	The number of *bytes* to create.
start	The first value to use in making the filter. The remaining values will follow consecutively from this one.
num•zeroes	The number of consecutive zeroes which are allowed in each valid byte in the filter.
bit•length	The total number of bits in each byte, normally between 8 and 10.

MASK PROCEDURE

Purpose: Sets and clears bits in a range of memory.

Syntax: **MASK(start,length,or•value,and•value)**

Remarks:

«start» The name of the variable which determines the start of the area to mask.

length The number of *bytes* to mask.

or•value The value to logically OR all bytes in the specified range with.

and•value The value to logically AND all bytes in the specified range with.

This procedure can set bits in a range with the **or•value**, and it can clear bits with the **and•value**.

Example: The following statement:

```
MASK(RBUF[300],500,$80,$FE)
```

Would set bit 7 and clear bit 0 of the bytes from the 300th element of **RBUF** to the 800th element of **RBUF**.

NEW COMMAND

Purpose: Erases the current program and clears the current variable space.

Syntax: **NEW**

NOT FUNCTION

Purpose: Returns the logical inverse of a value.

Syntax: **variable= NOT(expression)**

Remarks: If **expression** is 0, 1 is returned. In all other cases, 0 is returned.

NORMAL STATEMENT

Purpose: Returns text display to normal (non-inverse) mode.

Syntax: **NORMAL**

Example: See **FLASH**.

PACK PROCEDURE

Purpose: Places a text string into a byte array.

Syntax: **PACK name WITH "text"**

Remarks: **«name»** The name of the byte array where the string will be placed.

text A series of ASCII characters.

This procedure will place each character in **text** into the variable **name** and place an extra 0 at the end of **name** to terminate the string.

Example: The following program:

```
DEFINE BYTE[40] CAN
PACK CAN WITH "SOUP OR OIL OR LIME COLA"
PRINT !CAN
```

Would print **SOUP OR OIL OR LIME COLA** on the screen since that string was placed into **CAN** with the **PACK** statement.

PDL FUNCTION

Purpose: Reads a game paddle.

Syntax: **variable= PDL(expression)**

Remarks: **expression** can be a value from 0 to 3 and specifies the paddle to be read.

The returned value will be from 0 to 255.

On the Apple //c, only values of 0 and 1 should be specified for **expression**.

Example: The following program:

```
WHILE MEMORY[$C000]<128
   GOTOXY(10,10)
   PRINT PDL(0);"  "
ENDWHILE
MEMORY[$C010]=0
```

Would print the value of paddle 0 until a key is pressed.

In this example reference is made to location $C000, which is the Apple's keyboard address. This causes the program to loop until this location is greater than 127, which means that a key has been pressed. The access to location $C010 is made to clear the key that was pressed.

PLOT PROCEDURE

Purpose: Plots a dot on the lores screen.

Syntax: **PLOT(x,y)**

Remarks: **x** The x coordinate, from 0 to 39.

y The y coordinate, from 0 to 47.

The dot is plotted using the current color specified with the **COLOR** command.

Example: The following program would draw a blue 'X' on the screen:

```
DEFINE INTEGER POS
LORES
COLOR=2
WHILE POS<=39
   PLOT(POS,POS)
   PLOT(POS,39-POS)
   POS=POS+1
ENDWHILE
```

PR# COMMAND

Purpose: Sends all text output to a peripheral slot.

Syntax: **PR# expression**

Remarks: **expression** may be in the range of 0 to 7 and specifies the peripheral slot for output. 0 selects the normal video output.

Example: The statement:

```
PR# 1
```

Would direct all output to slot 1, which could be a printer or other peripheral device.

PRBLOCK PROCEDURE

Purpose: Displays a section of memory in a variety of formats.

Syntax: **PRBLOCK(start,length,label,digits,format,count1,count2,space)**

Remarks:

«start» The variable name specifying the first location to display.

length The number of *bytes* to display.

label The first number to show along the left margin of the displayed data.

digits The number of digits to show of the label above. This may be from 0 to 4. A dash is printed after the label, and a space can be printed after the dash if 8 is added to the value of **digs**.

format Selects how the data will be shown:
- 0= ASCII
- 1= HEX
- 2= HEX with bytes less than $80 being shown in inverse with $80 added to them.

count1 The number of bytes to display per displayed line.

count2 The number of bytes to display per grouping on a line.

space The number of bytes to display between groupings on a line.

Example: The following program:

```
DEFINE INTEGER TRACK,SECTOR,COUNT,ERR
DEFINE BYTE[255] BUFFER
WHILE TRACK<=34
   SECTOR=0
   WHILE SECTOR<=15
      COUNT=1
      RSECT(BUFFER,TRACK,0,SECTOR,COUNT,6,1,ERR)
      PRBLOCK(BUFFER,256,0,2,1,16,4,1)
      SECTOR=SECTOR+1
   ENDWHILE
   TRACK=TRACK+1
ENDWHILE
```

Would print out all of the data on the diskette in slot six, drive one.

PRINT PROCEDURE

Purpose: To display data on the screen.

Syntax: **PRINT (expression) (,) (;) (expression) (,) (;)...**

Remarks: **expression** may be on of the following:

A number, variable, or mathematical expression will print as a decimal value.

A byte array name with an exclamation point (!) in front of it will print as the text contained in it, up to the first zero in the array.

3) Text enclosed in double quotes.

Each expression may be followed by a comma or semicolon. A comma will cause the next printed expression to be at the next 8 position tab stop. A semicolon will cause the next printed expression to appear immediately following the previous.

A **PRINT** statement may be ended with a comma or semicolon, which will cause the next item printed to obey the rules stated above.

Example: The following program:

```
DEFINE BYTE CH
CH=32
WHILE CH<128
   PRINT CH,
   PRINTBYTE CH,
   PRINTHEX CH,
   PRINT !CH
   CH=CH+1
ENDWHILE
```

Would print the numbers from 32 to 127 in decimal, byte, hexadecimal and ASCII formats on the screen.

READ FUNCTION

Purpose: Reads a character from the keyboard with a cursor.

Syntax: **variable= READ(expression)**

Remarks: **expression** A number, from 1 to 4, which specifies the width of the cursor which is displayed.

This function will display a variable width cursor, and return the value of the first character which is typed on the keyboard. It is the normal way to read single character responses from within a program.

PRINTBYTE STATEMENT

Purpose: Prints 8 bit hexadecimal values.

Syntax: **PRINTBYTE (expression) (,) (;) (expression) (,) (;)...**

Remarks: This statement operates in the same manner as **PRINT** above, except that values print as two hex digits representing the lower byte of the value in hexadecimal.

Example: See **PRINT**.

PRINTHEX STATEMENT

Purpose: Print 16 bit hexadecimal values.

Syntax: **PRINTHEX (expression) (,) (;) (expression) (,) (;)...**

Remarks: This statement operates in the same manner as **PRINT** above, except that values print as four hex digits representing the value in hexadecimal.

Example: See **PRINT**.

PROCEDURE STATEMENT

Purpose: Defines a user subroutine.

Syntax: **PROCEDURE name**

Remarks: Chapter 6 has full details on the use of procedures and functions from within a program.

RECAL PROCEDURE

Moves the read/write head in the specified disk drive to track 0.

Syntax: **RECAL(slot,drive)**

Remarks: **slot** The slot number to use.

drive The drive number to use.

This procedure moves the head to track zero from whatever track it happens to be on by stepping the head out until it hits the track zero stop, guaranteeing that the head is at track zero.

Example: The statement:

```
RECAL(6,1)
```

Would move the read head on slot six, drive one to track 0.

RENAME COMMAND

Purpose: Changes the name of a file on a data diskette.

Syntax: **RENAME oldname,newname**

Remarks: The names specified may be either up to twelve character names in quotes, or the names of byte arrays containing the file names.

Example: The statement:

```
RENAME "MYPROGRAM","YOURPROGRAM"
```

Would change the name of the file **MYPROGRAM** to **YOURPROGRAM**. The program itself would not be changed.

RBLOCK PROCEDURE

Purpose: Reads one or more blocks from a ProDOS format diskette.

Syntax: RBLOCK(address,block,count,slot,drive,error)

Remarks:

«address» The name of the variable specifying the load address for the first block read.

«block» The name of the variable containing the first block number to read. On exit this variable will contain the last block number read.

«count» The name of the variable containing the number of 512 byte blocks to read in. On exit this variable will contain zero if no errors occurred, or the number of blocks left to be read if an error occurred.

slot The slot number of the disk drive to use.

drive The drive number to use.

«error» The name of the variable to place the return error code into. A value of zero signifies no error.

This procedure reads blocks from a ProDOS format diskette. This means 512 byte blocks, 8 per track, 280 per disk. This is the format used by Apple Pascal and NADOL's data diskettes.

Example: The following program would display all of the data on the diskette in slot six, drive one on the screen in ASCII format:

```
DEFINE INTEGER BLOCK,COUNT,ERR
WHILE BLOCK<=279
   COUNT=1
   RBLOCK(RBUF,BLOCK,COUNT,6,1,ERR)
   PRBLOCK(RBUF,512,0,2,0,32,32,0)
   PRINT
   BLOCK=BLOCK+1
ENDWHILE
```

RESULT

Statement

Purpose: Evaluates the expression to be returned as the result of a user defined function.

Syntax: **RESULT= expression**

Remarks: **expression** is evaluated and returned to the calling statement in the main program.

Example: See section 6 for full details on **FUNCTION** and **RESULT**.

RSYNC PROCEDURE

Purpose: Functionally identical to RTRACK, except that a reference mark on track zero is checked before reading.

Syntax: RSYNC(«pattern»,«address»,track,half,slot,drive)

Remarks:

«pattern»	The name of the byte array which contains the pattern to be checked before seeking and reading.
«address»	The name of the variable which specifies the address to begin reading at.
track	The number of the track to read from.
half	This is a zero or one signifying whether the half track should be read or not. Zero is the normal value and causes the normal track to be read. A value of one causes the next half track to be read instead.
slot	The slot number of the disk drive to use.
drive	The drive number to use.

Remarks: Before reading, this procedure seeks to track 0, finds the specified pattern, and then immediately seeks to the specified track and begins to read.

RSECT PROCEDURE

Purpose: Reads one or more sectors from a diskette.

Syntax: **RSECT(address,track,half,sector,count,slot,drive,error)**

Remarks:

«address» The name of the variable specifying the address to begin reading at.

«track» The variable containing the track number to begin reading on. On exit this will contain the track number of the last read sector.

half This is a zero or one signifying whether the half track should be read or not. zero is the normal value and causes the normal track to be read. One causes the next half track to be read instead.

«sector» The variable containing the sector number to start reading at. On exit this will contain the sector number of the last sector read.

«count» The variable containing the number of sectors to read. Sectors will be read in increasing order, moving to the next track when the highest sector number is reached. On exit this will contain zero if no errors occurred, or the number of sectors that were not read in if an error occurred.

slot The slot number of the disk drive to use.

drive The drive number to use.

«error» The name of the variable that the error return code should be placed into. This will be a zero if no errors occurred.

This routine reads using the format set with the **SETFORMAT** procedure.

Example: See **PRBLOCK**.

RTRACK PROCEDURE

Purpose: Reads the raw data from the specified track into the read buffer.

Syntax: **RTRACK(address,track,half,slot,drive)**

Remarks:

«address» The name of the variable which determines the starting location for the data read from the diskette.

track The track number to use.

half The half-track flag. A value of '0' causes the normal track to be read. A value of '1' causes the next half-track to be read.

slot The slot number of the drive to read.

drive The number of the drive to read.

RTRACK begins reading at the **address** specified above, and always ends at **RBUF[$3FFF]**. This is the ending address in **RBUF**, the main read buffer.

Data read from the disk has seven valid bits, 0-6 (the lower seven). Bit 7 (MSB) signifies whether the byte was a SYNC byte or not. If bit 7 is high, the byte was a normal 8 bit byte. If bit 7 is low the byte was followed by one or two zero bits on the disk, making it a SYNC byte.

Example: The statement:

```
RTRACK(RBUF,5,0,6,1)
```

Would read the raw data from track 5 of the diskette in slot six, drive one into memory starting at the beginning of **RBUF** and ending at the end of **RBUF (RBUF[$3FFF])**.

RUN COMMAND

Purpose: Begins the execution of a user program.

Syntax: **RUN**

Remarks: The **RUN** command clears the screen and positions the cursor in the upper left hand corner of the screen prior to executing the user program.

SAVE COMMAND

Purpose: Stores programs or data on a diskette.

Syntax: **SAVE filename (AT address,length)**

Remarks: **filename** A quoted filename of up to twelve characters or the name of the byte array containing the file name.

«address» If specified, this is the name of the variable which determines the location to begin the save at.

length If specified, this is the number of *bytes* to save to the diskette.

If the AT section is not specified, then the current program in memory is saved under the specified name on the current work disk.

The AT option allows data to be saved, such as hires pictures or blocks of information for future reference.

Example: The statement:

```
SAVE "PROGRAM"
```

Would save the current NADOL program in memory under the name **PROGRAM** on the current workdrive.

The statement:

```
SAVE "PICTURE" AT MEMORY[$4000],$2000
```

Would save the memory from $4000 to $5FFF in a disk file named **PICTURE**. This would save the hires picture buffer to disk.

SETFORMAT PROCEDURE

Purpose: Selects the format, address header, data header, and interleave for the RSECT, WSECT and FORMAT procedures.

Syntax: SETFORMAT(type,address•header,data•header,interleave)

Remarks:

type The type of encoding to be used on the diskette:
0= 6 and 2 (16 sector)
1= 5 and 3 (13 sector)
2= 4 and 4 (10 sector)

«address•header» The name of a byte array containing the six bytes to be used as the address header. The first three bytes are the address mark, and the next three are the closing mark.

«data•header« The name of a byte array containing the six bytes to be used as the data header. The first three bytes are the data mark, and the next three are the closing mark.

«interleave» The name of the byte array containing the interleave table for the diskette. See the section on pre-defined variables for information on built-in interleave tables.

This procedure sets up internal information used when reading and writing sectors.

Only the first two bytes of the closing mark are used for reading, the third byte is for writing only.

Example: The statement:

```
SETFORMAT(0,ADDR16,DATA16,INT16)
```

Would select 6 and 2 format with the standard address field, data field and interleave for normal DOS 3.3 diskettes. See section 9 for more information on built-in variables.

SIZEOF FUNCTION

Purpose: Determines the size of a NADOL disk file.

Syntax: **variable= SIZEOF(filename)**

Remarks: **filename** Up to twelve characters in quotes, or the name of the byte array containing the file name.

This function returns the number of bytes occupied on the current work disk by the specified file.

If the specified file does not exist, a value of -1 is returned, which allows NADOL programs to check for the existence of programs on a diskette.

STOP STATEMENT

Purpose: Terminates program execution and returns to immediate mode.

Syntax: **STOP**

Remarks: Setting **BREAK=1** will disable this statement.

TEXT STATEMENT

Purpose: Switches off graphic modes and returns to full screen text mode.

Syntax: **TEXT**

VLINE PROCEDURE

Purpose: Draws a vertical line on the lores screen.

Syntax: **VLINE(x1,y1,y2)**

Remarks: **x1** X coordinate of the starting point, 0 to 39.

y1 Y coordinate of the starting point, 0 to 47

y2 Y coordinate of the endpoint, 0 to 47

y2 must be greater than **y1**.

Example: See the **COLOR** statement.

WBLOCK PROCEDURE

Purpose: Writes one or more blocks to a ProDOS format diskette.

Syntax: **WBLOCK(address,block,count,slot,drive,error)**

Remarks:

«address» The name of the variable specifying the save address for the first block written.

«block» The name of the variable containing the first block number to write. On exit this variable will contain the last block number written.

«count» The name of the variable containing the number of 512 byte blocks to write out. On exit this variable will contain zero if no errors occurred, or the number of blocks left to be written in if an error occurred.

slot The slot number of the disk drive to use.

drive The drive number to use.

«error» The name of the variable to place the return error code into. A value of zero signifies that no error took place.

This procedure writes blocks to ProDOS format diskettes.. This means 512 byte blocks, 8 per track, 280 per disk. This is the format used by Apple Pascal and NADOL's data diskettes.

Example: **WBLOCK** operates in the same fashion as **RBLOCK**. See **RBLOCK** for a programming example.

WHILE/ENDWHILE STATEMENT

Purpose: Causes a section of a program to be executed repeatedly until a condition is met.

Syntax:

```
WHILE expression
   .
   .
   (executable statements)
   .
   .
ENDWHILE
```

Remarks: If the value of **expression** is zero then program execution resumes following the **ENDWHILE** statement.

If the value of **expression** is non-zero, then the statements up to the **ENDWHILE** are executed normally, and then control is transferred back to the **WHILE** statement where **expression** is tested again. Control will continue to be transferred back to the **WHILE** statement until **expression** evaluates to zero.

Example: See **COLOR, GOTOXY, HCOLOR, PDL, PLOT, PRBLOCK** and **RBLOCK** for programming examples.

WORKDRIVE STATEMENT

Purpose: Defines the drive for file operations.

Syntax: **WORKDRIVE slot,drive**

Remarks: **slot** The slot of the disk drive to use.

drive The drive number to use.

The **WORKDRIVE** statement selects the drive which will be used by all **LOAD, SAVE, DELETE, RENAME** and **INIT** commands.

Initially the **WORKDRIVE** is set to the boot drive.

Example: The statement:

```
WORKDRIVE 6,2
```

Would cause future file statements to use slot 6, drive 2 until another **WORKDRIVE** command was issued.

WSYNC PROCEDURE

Purpose: Functionally identical to WTRACK, except that a reference mark on track zero is checked before writing.

Syntax: **WSYNC(«pattern»,sync•size,pre•fill,track,half,slot,drive,«error»)**

Remarks:

«pattern» The name of the byte array which contains the pattern to be checked before seeking and writing.

sync•size A number, either 9 or 10, specifying the length, in bits, of each sync byte written.

pre•fill The number of byte times to wait before writing the data to the disk.

track The number of the track to write to.

half This is a zero or one signifying whether the half track should be written or not. Zero is the normal value and causes the normal track to be written. A value of one causes the next half track to be written instead.

slot The slot number of the disk drive to use.

drive The drive number to use.

«error» The name of the variable that the error return code should be placed into. This will be a zero if no errors occurred, a write protect error will be 255, a synchronization error will be 1.

Remarks: See WTRACK for the effect of the high bit of the data.

Before writing, this procedure seeks to track 0, finds the specified pattern, and then immediately seeks to the specified track, waits **pre•fill** byte periods, and begins to write. **Pre•fill** is used to let the disk drive turn a specific amount.

WSYNC always writes 10 SYNC bytes before the specified data.

WTRACK PROCEDURE

Purpose: Writes a section of raw data to a diskette from the write buffer.

Syntax: **WTRACK(sync•size,pre•fill,track,half,slot,drive,«error»)**

Remarks:

sync•size A number, either 9 or 10, specifying the length, in bits, of each sync byte written.

pre•fill The number of sync bytes to write out prior to writing any data. Normally this should be four or greater.

track The number of the track to write to.

half This is a zero or one signifying whether the half track should be written or not. Zero is the normal value and causes the normal track to be written. A value of one causes the next half track to be written instead.

slot The slot number of the disk drive to use.

drive The drive number to use.

«error» The name of the variable that the error return code should be placed into. This will be a zero if no errors occurred.

Remarks: This procedure starts at the beginning of WBUF and proceeds to write data until two consecutive zeros are encountered in the data (two consecutive zeros is not a legal sequence to write to the disk, so no conflicts arise).

The high bit in each byte determines how it will be written to the disk. If the high bit is clear, the byte is written normally. If the high bit is set, the byte is followed by extra zero bits, making it a SYNC byte on the disk.

WSECT PROCEDURE

Purpose: Writes one or more sectors to a diskette.

Syntax: **WSECT(address,track,half,sector,count,slot,drive,error)**

Remarks:

«address» The name of the variable specifying the address to begin writing at.

«track» The variable containing the track number to begin writing on. On exit this will contain the track number of the last sector written.

half This is a zero or one signifying whether the half track should be written or not. Zero is the normal value and causes the normal track to be written. A value of one causes the next half track to be written instead.

«sector» The variable containing the sector number to start writing at. On exit this will contain the sector number of the last sector written.

«count» The variable containing the number of sectors to write. Sectors will be written in increasing order, moving to the next track when the highest sector number is reached. On exit this will contain zero if no errors occurred, or the number of sectors that were not written in if an error occurred.

slot The slot number of the disk drive to use.

drive The drive number to use.

«error» The name of the variable that the error return code should be placed into. This will be a zero if no errors occurred.

Remarks: This routine writes using the format set with the **SETFORMAT** procedure.

Chapter 9

Built-in Variables

PREDEFINED VARIABLES

There are several variables which are built-in and therefore always available to a user program or from immediate mode. These variables are defined for some of the more commonly needed arrays, as well as for system variables which need to be accessible from a program. They are as follows:

MEMORY

Byte Array

Description: An array which encompasses the entire machine memory range. MEMORY[0] is location $0 in the computer's memory and MEMORY[$FFFF] is location $FFFF. All addresses in between work the same way. This allows access to any area in the machine's memory. Use caution with this array!

BREAK

Byte

Description: This byte controls the operation of ctrl-C. If set to a zero (normal) then ctrl-C will stop program execution. If set to a one, ctrl-C will not halt program execution.

If This value is set to a one then **<RESET>** will not return to immediate mode, it will cause the current program to start executing from the beginning.

Caution: If this variable is set to 1, the program running cannot be stopped by **<RESET>** or ctrl-C! This is a potentially dangerous situation since the computer must be turned off to return to immediate mode, erasing any changes made since the last save. To avoid this, only set BREAK to 1 in well tested code, possibly adding an exit command which sets BREAK back to 0 and ends the program.

ERROR Byte

Description: This byte contains the number of any error which has occurred. If this number is set to other than 0 it will be as though that error condition had occurred.

PRTSLOT Byte

Description: This byte contains the slot number to use for screen print operations (ctrl-P). It can be in the range of 1 through 7.

MACHID Byte

Description: Indicates the type of machine being used. This byte is set at boot time. The values for different machines are:

0= Apple][2= Apple //e
1= Apple][+ 3= Apple //c

If a non-Apple computer is being used which does not match any known Apple ID pattern, NADOL will assume that the computer is Apple][+ compatible.

HASLC Byte

Description: Indicates the presence of a language card in slot 0 of the machine. A zero signifies no card, a one indicates that one is available. This check is performed at boot time.

HASAUX Byte

Description: Indicates the presence of useable auxiliary memory in the machine. A zero signifies no card, a one indicates that auxiliary memory is available. This check is done at boot time. Note that this is always false on an Apple][or][+, and always true on a //c.

RBUF

Byte array

Description: This is the read buffer used by NADOL for raw data reads (**RTRACK**). It is $3FFF bytes in length.

WBUF

Byte Array

Description: This is the write buffer used by the NADOL raw data write routine (**WTRACK**). It is $29FF bytes in length.

ADDR16

Byte array

Description: This array contains the normal address mark for 16 sector diskettes. It is six bytes in length and contains the following values:

D5 AA 96 DE AA EB

This is normally used with the **SETFORMAT** procedure.

DATA16

Byte array

Description: This array contains the normal data mark for 16 sector diskettes. It is six bytes in length and contains the following values:

D5 AA AD DE AA EB

This is normally used with the **SETFORMAT** procedure.

ADDR13

Byte array

Description: This array contains the normal address mark for 13 sector diskettes. It is six bytes in length and contains the following values:

D5 AA B5 DE AA EB

This is normally used with the **SETFORMAT** procedure.

DATA13

Byte array

Description: This array contains the normal data mark for 13 sector diskettes. It is six bytes in length and contains the following values:

D5 AA AD DE AA EB

This is normally used with the **SETFORMAT** procedure.

FORM16

Byte array

Description: This array contains the numbers for the sectors on a normally interleaved 16 sector diskette. It is sixteen bytes long and contains the following values:

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

This is normally used with the **FORMAT** procedure.

FORM13

Byte array

Description: This array contains the numbers for the sectors on a normally interleaved 13 sector diskette. It is thirteen bytes long and contains the following values:

00 0A 07 04 01 0B 08 05 02 0C 09 06 03

This is normally used with the **FORMAT** procedure.

INT16

Byte array

Description: This array contains the normal interleave table for a DOS 3.3 diskette. It is sixteen bytes in length and contains the following values:

00 0D 0B 09 07 05 03 01 0E 0C 0A 08 06 04 02 0F

This is normally used with **SETFORMAT**.

INTPAS

Byte array

Description: This array contains the normal interleave table for a Pascal or ProDOS diskette. It is sixteen bytes long and contains the following values:

00 02 04 06 08 0A 0C 0E 01 03 05 07 09 0B 0D 0F

This is normally used with **SETFORMAT**.

INTCPM

Byte array

Description: This array contains the normal interleave table for Apple CP/M. It is sixteen bytes long and contains the following values:

00 03 06 09 0C 0F 02 05 08 0B 0E 01 04 07 0A 0D

This is normally used with **SETFORMAT**.

INT13

Byte array

Description: This array is the interleave table for diskettes which do not use software interleaving, such as 13 sector disks. It is thirteen bytes long and contains the values:

00 01 02 03 04 05 06 07 08 09 0A 0B 0C

This is normally used with **SETFORMAT**.

WNDLEFT

Byte

Description: The value for the left margin of the text screen. Normally set to 0. The sum of this value and WNDWIDTH should never exceed 40.

WNDWIDTH

Byte

Description: The value for the width of the text screen. Normally set to 40. The sum of this value and WNDLEFT should never exceed 40.

WNDTOP Byte

Description: The value for the upper line of the text screen. Normally set to 0. This value should be from 0 to 22, but never greater than WNDBOT.

WNDBOT Byte

Description: The value for the lower line of the text screen. Normally set to 23. This value should be from 0 to 23, but never less than WNDTOP.

The NADOL master disk is shipped with Nibbles Away III and several other utilities. This section discusses the use of these utilities in detail.

When the NADOL master disk is booted, first a copyright message will appear on the screen. Then the NADOL logo will be shown. At this point, pressing the <SPACE> bar will cause the main NAIII menu to be displayed. It will look like this:

```
NADOL        NIBBLES AWAY III        VERSION
00000                                  1.0

                  MASTER MENU


            1 - NIBBLE COPY
            2 - FAST SECTOR COPY

            3 - TRACK EDITOR
            4 - SECTOR EDITOR

            5 - DISK SPEED TEST
            6 - CONFIGURATION
            7 - AUTO EXECUTE

            8 - EXIT TO NADOL

            9 - USER APPLICATIONS

      ENTER SELECTION ->0          ? FOR HELP
```

The '?' key will cause a HELP screen to be displayed. This key is active in each of the options on the menu, and can be used to obtain useful information. For example, in the sector editor, the question mark will display a list of valid commands and their functions.

Each of the other options on this menu will be discussed separately, since each pertains to a different utility program. For now we will talk about the last two options, #8 and #9.

Pressing '8' from this menu will exit to the NADOL immediate mode. If this option is pressed by mistake, simply type 'RUN' and press <RETURN> to show the main NAIII menu once again (a message on the screen will remind you of this). The immediate mode can be identified by the dot prompt '.', which shows at the left hand margin. From this mode, all of the features of NADOL are directly accessible, as discussed in section 1 of this manual. (For those users who wish to create NADOL work disks without NAIII on them, see the note at the end of this chapter).

Option '9' switches the screen to a list of 'user applications'. These are programs written by the user which can be run directly from the main menu. There will be several suggestions in 'Nibble News', the Nibbles Away newsletter, available from COMPUTER:applications, Inc. As shipped, there are no names on this list, but any desired names may be added by editing the 'NAIII' program using the NADOL editor, described in section 1.

On the following pages each of the other options will be discussed, one at a time.

Printing the screen

At any time, control-P (press and hold the control key and then press 'P') will cause the screen to be printed to the printer. NADOL is shipped to use slot1for a printer, but this may be changed using option #6 (configuration, see below). As the screen is printed, an underline will pass over the screen, showing the progress of the printout. When the printout is complete, NADOL (or NAIII) will resume what it was doing.

NIBBLE COPY

The Nibble copy option (#1) is the normal way to copy most protected diskettes. It has several different options which enable it to duplicate just about any protection system currently in use.

When this function is chosen, a selection screen will be displayed with several options. Each option has a number to the left of it which is used to change that particular option. If an option cannot be changed (i.e. the slot number with only one disk controller in the system), then that option will have its number replaced with an asterisk (*) to show that it may not be selected. Each option has the following function:

1. Changes the source slot. Only slots which contain disk controllers will be displayed.
2. Changes the source drive from 1 to 2 and back again.
3. Changes the destination slot.
4. Changes the destination drive.
5. Changes the starting track number. When this option is selected, the cursor will be positioned to the right of the start track line. Then a starting track value between 0 and $23 may be entered (this number is entered in HEX). If a half-track value is desired, add '.5' to the number.
6. Changes the ending track number. This is entered in the same fashion as the starting track, above. The only restriction is that the ending track must be larger than, or equal to, the starting track.
7. This option specifies by what increment the copy will step for each new track. It is entered in the same way as the previous two options. The only restriction is that the step be greater than 0 and less than $23.
8. This toggles the Synchronization flag. When set to '[ON]', each track on the destination disk will be placed in the same physical orientation as on the original. Normally this option should be set to '[OFF]'. See section 3 for more details on synchronization.

9. This toggles the Nibble count flag. When set to '[ON]', each track on the destination disk will have the exact same number of nibbles as the corresponding track on the original disk. Normally, this should be set to '[OFF]'. See section 3 for more details on Nibble counting.

0. This option will show another screen of options. These options are those which do not normally need to be modified. This second screen operates in the same way as the previous screen. The options allow a specific address mark to be looked for, as opposed to the automatic address mark selector. Also, the default sync byte size of 10 bits may be set to 9 bits if desired. Normally, these options should be left in their default condition.

Normally, these options will already be set correctly. The only item which normally has to be changed is the source/destination information (options 1-4), if a single drive system is being used.

CAUTION: Original diskettes should ALWAYS have a write protect tab on them when performing a copy, especially on a one drive system where the disks are being exchanged into the disk drive. This will eliminate the possibility of destroying a master disk due to insertion in the wrong drive!

After all of the options are correct, press the <RETURN> key to begin the copy. A prompt will ask for both disks to be inserted if using two drives, or for the source disk to be inserted if only one drive is being used. After inserting the appropriate disks, press the <RETURN> key to continue. As the copy progresses, a status display will appear on the screen. The result of the copy on each track will be displayed as it is completed.

The status code is a three digit sequence. The first character is a 'Y' or 'N', indicating whether or not that particular track was successfully copied. The second number indicates the number of read errors encountered, while the third shows the number of write errors which occurred on that particular track.

On any given track, NAIII will give up if either five read errors or five write errors occur. Read errors normally mean that NAIII is unable to locate any valid data on a particular track. Sometimes this may simply indicate that the track has no usable data on it, other times it may indicate a very advanced protection scheme. Write errors indicate that there was trouble putting the data on the destination diskette properly. This could mean that the drive speed is off, or that there is no disk in the drive.

Generally, if the first character in the status code is 'Y', then the track was properly copied, regardless of what the two errors counts indicate.

During the copy, several keys can be pressed to invoke special functions. They are as follows:

Q - Aborts the current copy process.
S - Skips to the next track.
G - Enables the graphic display mode (mainly for entertainment).
T - Disables the graphic mode, and switches back to text mode.

There are two irrecoverable errors which may occur during a copy:

WRITE PROTECT ERROR
The destination disk has a write protect tab on it.

UNABLE TO SYNCHRONIZE ERROR
This will normally only occur if trying to perform a synchronized copy on a disk with no locatable data on it (i.e. a blank disk).

Both of these will stop the copy in progress. Check the diskettes and retry the copy.

FAST SECTOR COPY

This option is used for diskettes which are 16 sector disks. This includes all DOS 3.3, Apple Pascal, and Apple CP/M diskettes. This option is much faster than the bit copy mode, and since it knows exactly what type of disk is being copied, it can usually do it more accurately.

Another benefit is that a language card, and/or the auxiliary card in a //e or //c will automatically be used (if both are present, only two disk swaps are necessary to copy a whole disk!).

When this option is chosen, a selection menu is shown. The options are the slot and drive for the source and destination disks. Each option has a number to its left. If only one disk controller exists, then both slot options will show and asterisk (*) instead of a number, since that item may not be changed.

Press the desired options until the proper slots/drives are shown. Then press the <RETURN> key. The screen will then display a message asking for the source (and destination on a two drive system) disks to be inserted. After inserting the disk(s), press <RETURN> to begin. The copy process will switch between the two disks, showing the current action on the screen in inverse. If a one drive system is being used, the program will stop and ask for the appropriate diskettes when necessary.

After the copy is complete, the program will ask if another disk should be copied. If 'Y' is answered, the option screen will be displayed and the process can be repeated as often as desired. If 'N' is entered, the main NAIII menu will be shown. (The master disk should be inserted into the drive before selecting 'N', or the program will ask for it).

Track Editor

This option is used to view the raw data present on any track on a diskette. Several options are available to allow the data to be scanned or modified, and then written back to the disk if desired.

When first selected, this function will display several hundred bytes of information from the start of the read buffer on the screen. A blinking cursor will appear in the upper left hand area of data. This is the current location pointer. Its location is displayed in the top left corner of the screen at all times. In the upper right corner is the current track number, initially set to zero.

Different keys from the keyboard invoke different functions. The '?' key will display an abbreviated list of all of the commands and their functions. The full list is given below:

I..... Moves the cursor up one line.
J..... Moves the cursor on byte to the left.
K..... Moves the cursor on byte to the right.
M..... Moves the cursor down one line.
Arrows.... Perform the same functions as the keys listed above.
>..... Moves the cursor down one page.
<..... Moves the cursor up one page.
+..... Increments the current track number.
-..... Decrements the current track number.
T..... Asks for a new track number to be entered from the keyboard.
O..... Shows the Options page which allows the selection of different slot/drive combinations.
F..... Asks for a HEX string to find. The cursor is moved to that location if the string is found.
C..... Shows the count of the number of bytes to the next occurrence of the bytes at the cursor location.
P..... Shows the print menu, asking for a start and end location, and a title for the data to be sent to the printer.
S..... Sets the 'Track Start' value to the cursor location.
E..... Sets the 'Track End' value to the cursor location.
M..... Moves the currently marked track (start-end) into the write buffer and sets the 'write buffer end' value accordingly.
/..... Toggles between read and write buffer display.
G..... Prompts for a new location for the cursor.

R..... Reads the current track into the read buffer.
W..... Writes the data in the write buffer to disk.
Q..... Quits the track editor.
Space.... Enters the modify mode. Once entered:
Typing HEX values will change the value of the current byte.
The space bar will move to the next value.
The <RETURN> key will accept the current value.
The arrow keys may be used to move the cursor.
The <ESC> key will abort the modify.

Data which has its high bit clear will be written to the disk as a SYNC byte, and will display on the screen in inverse.

The main use for the track editor will be to examine the data on the various tracks on a diskette to determine where the data lies, and what it is composed of. The data seen with this module is the 'raw' disk data. To view the decoded version of the data, the sector editor should be used.

Sector Editor

Option #4 invokes the sector editor. This module is used to view the data on a disk which is either 16 sector, or 13 sector compatible. This includes all DOS 3.3, DOS 3.2, Apple Pascal and Apple CP/M diskettes. Along with the ability to view the HEX and ASCII representation of the data, a disassembly of the data may be viewed.

When this module is chosen, the standard display will be shown on the screen. This consists of a HEX display of the current sector to the left, and an ASCII display to the right. At the top of the screen, the current track and sector are displayed.

There are a number of commands available, each invoked with a different key from the keyboard. As usual, the '?' key will show an abbreviated list of commands. This is the full list:

I..... Moves the cursor up one line.
J..... Moves the cursor one location to the left.
K..... Moves the cursor one location to the right.
M..... Moves the cursor down one line.
Arrows... Perform the same functions as the keys above.
+..... Increments the sector number. If the sector number becomes greater than the number of sectors on the track, the track number is incremented by one, and the sector number is set to zero.
-..... Decrements the current sector number.
>..... Increments the current track number.
<..... Decrements the current track number.
/..... Toggles the cursor between the HEX and ASCII displays.
Space... Enters the Modify mode:
HEX digits are accepted on the HEX side, space move to the next value.
On the ASCII side, any characters may be typed.
The arrow keys will move the cursor around.
The <RETURN> key accepts the current change and exits the modify mode.
The <ESC> key aborts the modify mode.
R..... Reads the current sector. If an error occurs, the ERROR message is displayed, but whatever partial data which may have been read will still be shown in the data displays.

W..... Writes the current sector.
D..... Disassembles the data in the buffer from the cursor location.
The forward and backward arrows page through the listing.
The <RETURN> key exits the disassembly mode.
T..... Prompts for a new track number, in HEX.
S..... Prompts for a new sector number, in HEX.
F..... Selects the Find mode.
The find menu is displayed and the following keys are active:
1.... Prompts for the starting track.
2.... Prompts for the starting sector.
3.... Prompts for the ending track.
4.... Prompts for the ending sector.
5.... Switches between HEX and ASCII searching.
6.... Switches the search direction between ascending and descending. This specifies whether the search will progress downward through the sectors on each track, or upward through them.
7.... Prompts for a string to search for, up to 32 characters.
Pressing <RETURN> starts the search.
Pressing 'Q' aborts the search.
O..... Displays the Options menu. The following keys are active:
F.... Switches the format between 13 and 16 sector.
I..... Switches the interleave between the various types.
H.... Switches the High bit setting for ASCII characters entered in the modify mode.
<RETURN> accepts the current selection.
Q..... Quits the sector editor.

The sector editor can be used to look at the data on a diskette, allowing it to be changed easily. This means that DOS commands or the Pascal system prompts may be changed at will on a diskette.

If the sector editor is unable to read a particular sector, it usually means that there is something wrong with the diskette. It may be, however, that the protection scheme on a given disk prevents a sector from being read. To see what the raw data on a disk looks like, the track editor can always read any given track and display what its underlying format is.

Disk Speed Test

Option #5 starts the disk speed test utility. Using this utility, any disk drive can be set to the correct speed. Appendix C shows a diagram of the inside of an Apple disk drive, and the location of the speed control adjustment.

WARNING: **Disk speed adjustment requires that the disk drive be opened. If done improperly, this procedure can cause severe damage to the disk drive, and possibly the Apple. Excercise caution when performing this adjustment.**

To perform this function, the disk drive in question should be opened *before* the power is turned on. The drive will operate properly without its cover. Turn on the computer and select the disk speed test (#5).

The initial selection menu allows the slot and drive to be selected. Once this is complete, press <RETURN> to continue.

Next, a warning message will appear, explaining that the disk used for this test will be erased. At this point, insert a scratch diskette into the drive to be tested, and press <RETURN>.

The speed test ruler will appear, and a pointer with a number below it will begin to move back and forth below the ruler. The number is the current disk speed. Above the ruler are some statistics about the drive. These include the minimum and maximum speed so far, as well as the average speed so far. If the speed is very far off, the pointer will appear at the edge of the ruler, and a beep will sound at regular intervals. This indicates that the disk speed is out of tolerance.

The disk speed may be adjusted *slowly* while the test is running, and the results should be immediately apparent. The ideal speed for the disk drive is between 0 and -10. Some variation of the disk speed is perfectly normal. As long as the speed does not vary by more than 15 units between any two readings, the disk stability is fine.

Once the disk speed has been set, the computer should be turned off. Then the disk drive can be closed back up, and put back into normal service. It may be desirable to run this test periodically, since most disk drives have a tendency to speed up over time. This is normal and due simply to the aging of certain components on the circuit board in the disk drive.

The figure supplied in the appendix pertains to standard Apple drives. If your disk drive does not look like the one shown, it still may be possible to adjust the speed. There will usually be a separate motor speed control board within the drive, normally at the back of the drive. Some drives required that the baseplate be removed to access the speed adjust control, since it pointed downward. Some drive have the speed control circuitry included into the rest of the electronics on a single board. In this case there may be several controls which all look alike. The best option in this case is to ask the manufacturer, or your dealer.

A note on disk speed

If the speed of a disk drive is too high, there will be less time to write data to it on each revolution of the disk. This can cause problems during a copy, and is the main cause of write errors. If write errors occur, try slowing down the disk drive by 5 or 10 units and re-running the copy. Many times this will solve the problem. The disk controller circuitry can handle a good deal of variation in speed, so this change will not affect the normal operation of the drive.

Configuration

Option #6 starts the configuration module. This allows the selection of display which will be used by the NADOL program editor, and allows the default printer slot to be set.

This option will display a screen which shows the current printer slot and editor configuration. Pressing the 'P' key will change the printer slot. If no printer is installed, set this to 0. This will disable the screen print function.

Pressing 'E' will change the type of editor. When both of these options are at the desired values, press the <RETURN> key to make the changes permanent.

The next time that the NADOL disk is booted, these changes will take effect. If the changes are desired immediately, simply boot the disk now.

Auto-Execute

The Auto-execute section is provided for those programs which do not copy using the nibble-copy option.

When this option is selected, the slot and drive for the source and destination may be selected in a manner similar to that used in the nibble copy section. Then the desired Auto-execute procedure is selected from the menu with the arrow keys.

Pressing <RETURN> will start the auto-execute procedure. During this operation, the program may stop and ask for the source and destination disks to be inserted at various times. When complete, a message will be displayed and the disks may be removed. Another Auto-Execute may be performed by pressing <RETURN> to select another, or pressing 'Q' will return the user to the main NAIII menu.

Chapter 13

Disk Protection

Throughout the evolution of disk protection on the Apple, there are several techniques which have become popular. This chapter will examine each of the major protection systems, explaining how they work, how to detect them, and finally, how to bypass them to obtain back-ups.

In the process of designing the disk hardware for the Apple, particular attention was paid to making it software intensive, rather than hardware intensive. This means that a great deal of the burden for operating the disk drive is placed on the controlling software, instead of building larger and more complicated circuit boards to do the work. This aproach reduced the cost of the Apple disk hardware, and at the same time increased its reliability by cutting down the number of components on the board. When it was introduced, the Apple disk drive was considered an engineering marvel due to its advanced design.

Since control of the hardware is placed mainly on the software, there is a great deal of variation possible. This is what allows the huge variety of copy protection systems which are on the market today to exist. Any time a particular phase of the operation of the disk drive can be performed in more than one way, there is a good chance that it is used in a disk protection scheme somewhere.

First, we will examine the method used to store data on a diskette under a normal operating environment, DOS 3.3 in this case. The physical mechanics of the disk drive spin the diskette media inside its protective jacket at 300 RPM. A read/write head moves across the disk, from the outer edge to the inner one, much like the motion of the needle on a phonograph. This creates 'rings', called tracks, around the diskette where data can be stored.

Each track can hold up to a theoretical maximum of about 6000 bytes of information, and there are thirty five tracks per diskette. If we were to use each track to hold a single piece of information, they could be up to 6000 bytes long, but if they were smaller, the extra room would be wasted. Obviously, this would make a diskette unusable for general data storage.

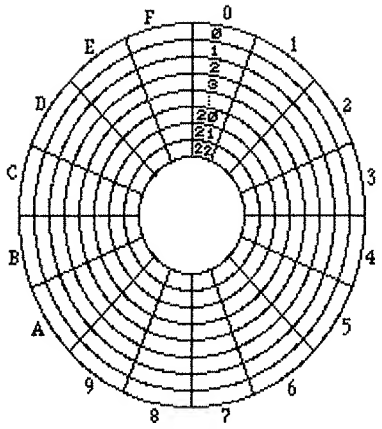


Figure 1
Diskette Layout

Instead each track is divided up into sixteen equally sized chunks called sectors. Each of these sectors can hold two hundred fifty six bytes of information. Figure one graphically illustrates how a diskette is physically laid out. This results in just over four thousand bytes per track. The reduction in storage is due in part to the extra information used to mark where each sector starts and stops, but mostly to the restrictions on the exact data which may be stored in a given byte (described in detail below). For now, we can assume that about four thousand (4k) bytes are available per track. The minimum amount of space that must be used to hold a chunk of information is only 256 bytes, which means that wastage is kept to a minimum.

In order to be able to locate a particular sector on a given track, special information is placed in front of each sector so that the read/write head can spot it as the diskette spins by. Figure two shows a step by step blow-up of the different sections which make up the data on the diskette. The address field contains four pieces of information about the sector which follows. Each one takes up two bytes in the address field due to a special type of encoding, described later.

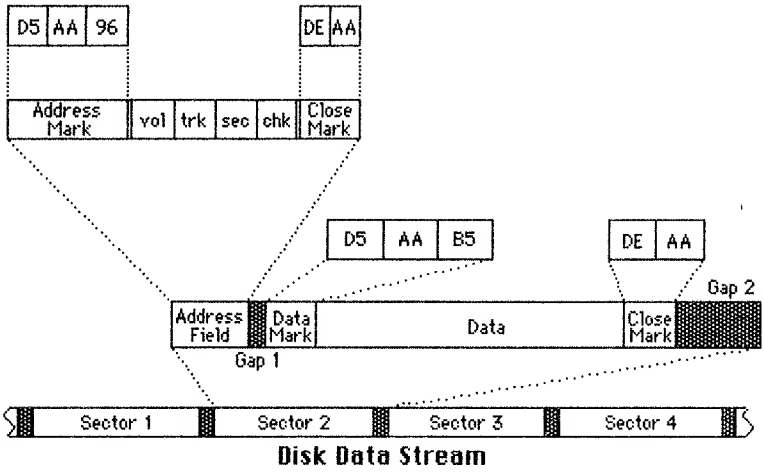


Figure 2
Blow-Up of Track Data

The first piece of information is the disk volume number. This was originally designed to be used to make sure that the diskette in the drive was the proper one. In fact, DOS 3.1 required that the proper volume number be specified for all commands except catalog. The problem with this scheme is that each disk must have a different number, but only two hundred fifty five volume numbers are available. This system is no longer in use, but the volume number lingers on to retain compatibility.

Next, is the track number on which this data resides. This may seem redundant, but it is used to verify that the read/write head is positioned over the proper track before reading or writing any data.

Following this is the sector number. The software controlling the disk drive reads consecutive address fields until it finds the one containing the desired sector number. At this point, the specified operation is performed.

The last is a code which is used as a checksum to verify that the previous three pieces of information were read properly.

The next topic to be considered is how the data actually appears on the diskette. The information on the diskette is actually just a series of magnetic field changes. Once the read/write head passes over them and the associated hardware does its job, the result is a series of 1's and 0's which are sent from the disk drive to the controller in the Apple.

Three restrictions are placed on the actual bytes which may be processed by the disk hardware:

1. Each disk byte must have its high bit set.
2. There may be no more than two consecutive zero bits in a byte.
3. Only one pair of consecutive zero bits may appear in a given byte.

Restriction 1 is used in conjunction with self-sync bytes to enable the disk drive to locate the start and end of the data on the drive. A self-sync byte is a byte which has all '1' bits ($FF) with extras 0's after it on the drive. In the case of DOS 3.3, there are two extra zeros after each self-sync byte. Since each byte must have its high bit set, this enables us to locate the proper start/stop of each byte following at least four self-sync bytes. Figure three shows that no matter where in the bit stream the hardware begins to read, after four self-sync bytes have passed by, the head will be properly in sync with the data.

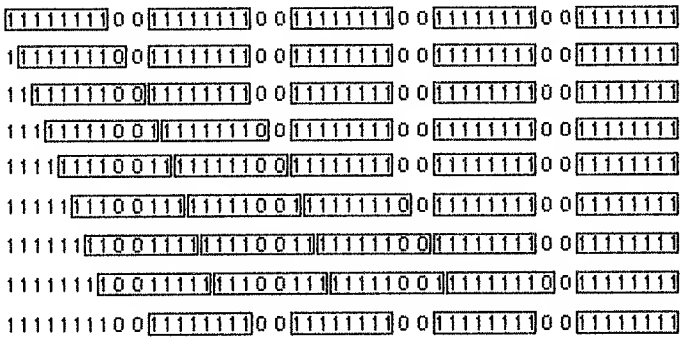


Figure 3
Function of SYNC bytes

Restriction 2 is caused by the hardware since a series of 0 bits means no change in the magnetic field on the diskette, and after 2 bit times, the head begins to return invalid data if no magnetic change occurs.

Restriction 3 is added for DOS 3.3 to eliminate a few values and leave 64 valid bytes. 64 possible bytes means that 6 bits of data can be written to the disk for each byte which goes out through the disk hardware.

Since we are restricted to 6 bits per byte, an encoding system must be used to allow us to place 256 bytes of data, each containing 8 valid bits, onto the diskette. The way that this is done is to split each byte into two groups, one of six bits, the other with two. The six bit group can be sent out directly, while the two bit group is taken in three's (making six bits) and written out. The total number of bytes which actually written is 342. This is the reason that the actual storage capacity of a track is lower than the theoretical capacity, as stated above.

This gives us a rudimentary description of the working of the disk drive and its control software. The protection systems that are currently in use all make use of some method by which they can modify one or more of the basics discussed above. By doing this, they prevent the normal disk controlling software from reading the diskette and thus, prevent duplication.

The various techniques currently in use will be discussed in the following chapters of this section. They will be presented roughly in the order in which they appeared on the market. This should give the reader some idea of the path that disk protection has taken over the last few years.

Modified Checksum

The checksum information found in the address field is computed by using a particular combination of exclusive-or's among the previous data. As long as all disks use this convention, no problems arise. Software manufacturers found that by modifying how the checksum was created, they could prevent normal duplication programs from reading their software. This type of protection started to be used on a large number of programs since it was very easy to implement, and very reliable.

The one flaw to this system was that it was very easy to get around. All that was necessary was to disable the checksum verification, since it is really only a precautionary measure. This allowed most normal duplication programs to read diskettes protected with this system. Most of the time the new diskette would contain all of the correct information, but with the checksums on the disk now generated by the standard method. This meant that the disk did not operate because the disk read/write code on the disk was set up for the modified checksum system. The solution to this was to also disable the checksum verification on the new disk. This resulted not only in a duplicate, but in a duplicate which was no longer protected!

Today's sophisticated 'nibble copy' programs do not use the checksum information on a diskette at all, so they are not bothered in the least by this type of protection. However, any diskettes created by a nibble copy program will retain the same protection as the original, since the protection is also reproduced.

Modified Address Mark

The next type of protection to be used was modified address marks. This involved changing the normal D5 AA 96 address mark to some other combination. This prevented normal backup programs from reading the disk since they were unable to locate the start of the sectors on the disk.

To combat this, it was necessary to change the address marks in the backup program being used. The problem was that every disk must have at least one normal sector on it in order to boot. This meant that the backup program had to be able to read at least two different formats. Because of this, one could not simply change the backup programs built-in address mark and begin to make a backup.

This problem was further complicated when manufacturers began to use several different address marks on a single diskette. At this time, it was no longer practical to use a normal backup program modified for different address marks. This meant that nibble copiers were needed to back-up this type of disk.

Nibble copiers have used several different methods for dealing with diskettes which have several different address marks on them. One way is to actually specify the address mark, but that requires a change for each different address mark used. Fortunately, most nibble copiers require only the address mark of one of the sectors on a given track. Many times this cut down on the number of different address marks used.

Another method was to look for the GAPs (explained previously) which precede each sector. This technique did not work for long, however. Manufacturers found that there were actually many different bytes which could be used in the GAPs instead of the normal FFs. This caused problems for many nibble copiers, but it had even greater effects in some cases.

Many times the bytes which were chosen as the replacement GAP bytes were not as stable from the controllers standpoint. This meant that the disk controller could not always find the correct data on the diskette properly. This resulted in higher failure rates on the diskettes, and the end result was that the customer was burdened with faulty software.

The latest method for determining the address marks used on a particular diskette is via the GAPs. This time however, it is not the value of the bytes within the GAPs which is used. The nibble copier looks for a section of SYNC bytes, regardless of their value, and uses this to determine where the sectors begin. This method can 'see through' many protection systems and has a very high reliability. Another bonus is that it can perform this analysis on most diskettes, making the backup process almost completely automated.

The main disadvantage to both the modified checksum and the modified address mark systems is compatibility. Data written by either one of these systems cannot be read back by a normal system. This means that a word processor or spreadsheet using this type of protection could read files which it created, but not those created by any other system. Usually this was sufficient for games, but business programs could not operate in this fashion. This made these systems less desirable, and hence they are used less and less today.

Synchronization

Manufacturers needed some type of system which they could trust, yet one which would allow perfectly normal data to be placed on a diskette. Synchronization provided this.

The layout of a diskette shown in figure one is actually an ideal condition. In real life, the sectors on each track do not line up with those on the next track in an orderly fashion. However, whatever the orientation of the tracks is, it will never change. Manufacturers used this fact to create a protection system.

When the software was created, the orientation between two tracks, or the spin angle (figure 2) was checked and recorded on the disk. Later when the user booted the disk, this angle was checked again. If it had varied, it meant that the data was no longer on the original disk and the software would not operate.

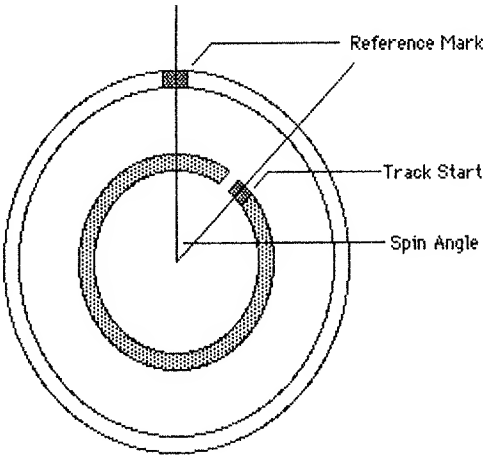


Figure 4
Spin Angle

This system worked very well because it allowed any data to be placed on the disk, and it did not use any odd or unreliable methods of storing information on the diskette. It also was beyond the power of all of the existing nibble copy programs when it was first used. That did not last long.

One of the telltale signs of this type of protection is a very fast seeking sound from the disk drive. This is created by the stepper motor as it moves the read/write head back and forth between the two tracks for which the spin angle is being checked.

In order to duplicate this protection system, it is necessary to correctly reproduce the spin angle for each track on the destination disk. This is done by first copying track zero and finding a convenient unique reference mark which can be reliably located. Before reading each track on the original, the reference mark is found. At this point, the head seeks immediately to the track being duplicated, and then the number of bytes which pass below the head before the first sector is encountered is recorded. The reference mark is then located on the destination drive. The head then moves to the destination track, and writing is suspended until the same number of bytes recorded during the read have passed under the head. This preserves the spin angle and properly duplicates this type of protection.

This system is still used today, since it is reliable and requires a nibble copier with advanced features in order to make a backup of it.

Nibble Counting

Every disk drive spins at a slightly different speed. When reading data, this is taken into account by the disk controller, so there is no perceivable effect to the user. On writing, however, the speed of the diskette affects the number of bytes which will fit around a particular track. The variation in this number will be small, but there will be a variation, since no electric motor turns at a perfectly constant speed.

The nibble counting protection system uses this fact. When a particular track is written out, the actual number of bytes which exist on the track is counted and recorded. Later, when the program boots, this count is checked. If the disk has been duplicated, the speed of the destination drive will be different than that of the disk which originally created the master. This means that there will be a few more or less GAP bytes on the disk. This small difference is enough for the nibble counting protection system to determine that the disk has been duplicated.

This system is very difficult for a nibble copier to duplicate. There are basically two methods which can be used. The first is the manual method. In this system the nibble copy program begins to write out the data over and over. Between each write, it counts the number of bytes which actually fit on the track. If the number is not correct, it tries again. During this time the user is told whether too many or too few bytes were recorded. Using this information, the user manually adjusts the speed of the disk drive. This system works very well, but it requires that the disk drive be opened to perform it properly, which can be rather cumbersome.

The second method can be done automatically by a program, but it is not as reliable. In this method, the non-essential data on the track is converted to SYNC bytes, a few at a time. The bytes which are made SYNC become 25% longer, and take up more room. This allows the track to be expanded under software control. The main drawback is that not all bytes may be turned into SYNC bytes reliably, which can cause some data problems. In most cases, however, this technique is quite adequate.

Sometimes nibble counting is combined with synchronization. This can be duplicated, but it can take a very long time, since these are the two most time consuming protection systems to back-up.

Spiral Tracks

One assumption which is normally made about data on a diskette is that information may not lie on consecutive half-tracks. This is because the read/write head's path is wider than one half track, so when one of the two half-tracks were written, it would obliterate the other. There is a variation on this theme, however, which has been used in a very effective protection system.

If a section of data only exists for half of a revolution of the diskette, then the other half could technically be used for data on a half track, without the conflict described above. This is exactly what is done in spiral tracking. Figure four shows this system graphically.

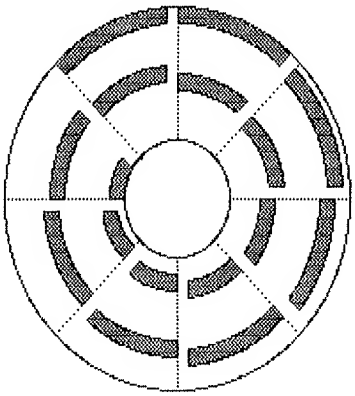


Figure 4
Spiral Tracks

The gray areas show where the data resides. Even though there is data on all of the half tracks, this system provides that no data will be *adjacent* to another half track containing data. The gaps of white between the sections of data are provided to allow the read/write head time to move from one track to another. This system is fairly reliable, but requires that the user's drive be able to step to the half tracks. This is not a problem for Apple drives, but some outside vendors have marketed disk drives which can only step to the normal tracks. These drives will not operate correctly with this type of protection.

Many times, this system is mixed with a modified address mark system. It may even incorporate a system wherein the address mark is different for each and every section of data on the disk. Usually, the end of a particular section of data will specify the address mark for the following section of data. This type of protection can have many permutations added to it, and is very difficult to back-up without some form of intelligence in the backup program (like that found in NADOL).

The procedure for backing up a disk with this type of protection is very similar to that used for synchronized protection. The difference is that instead of writing full tracks out, only small portions of tracks are written.

First track 0 is copied, and a reference mark is located. Then the tracks starting (usually) with 1 are duplicated, preserving the spin angle. As each track is written, only the actual section of the track which contains data is written. This keeps the data from bleeding over into the tracks which have previously been backed up.

Sometimes only a section of a particular disk will be protected in this fashion. Since this system wastes so much space, only small programs may be protected entirely in this fashion. Usually there will be the sound of very rapid, continuous read/write head movement during the time when this system is being used. The user may listen for this while a diskette is booting to determine if this type of protection is in use.

Hidden Sync Bytes

One type of protection which was very effective, but is now basically obsolete, is hidden sync bytes. This system involved placing one or more sync bytes (those which are normally only used to lock the controller) in strategic positions. When this system was first introduced, there was no easy way to detect the presence of sync bytes. This meant that a sync byte could be present but there was almost no way to locate it. The only effective way was to actually take apart the code which identified the SYNC byte on the disk.

Today, however, things have changed a bit. NADOL can distinguish between sync and non-sync bytes as it reads the disk. This means that all sync bytes will be preserved in their proper locations, normally without any user intervention whatsoever. This has made this type of protection much less popular today. However, it is still used on some disks.

Appendix A

Error messages

ERROR MESSAGES

The following table is a list of all of the error messages and their associated numbers. The numbers are those that will be returned in the variable '**ERROR**'. If **ERROR** is set to one of these numbers, the appropriate error message will be displayed as if the error had really occurred.

1 = SYNTAX ERROR

Occurs when NADOL cannot understand a command. Usually the result of spelling errors.

2 = ( ) MISMATCH ERROR

The open '(' and close ')' parentheses in an expression do not match.

3 = PARAMETER COUNT ERROR

When calling a built-in procedure or function, an incorrect number of parameters was entered.

4 = STACK OVERFLOW ERROR

The internal stack cannot hold any more information. Usually caused by one of the following:

Nesting **IF-ELSE-ENDIF** or **WHILE-ENDWHILE** statements too deeply.

Calling user defined procedure or functions from within each other to more than 11 levels.

Nesting parentheses to more than 16 levels.

5 = DUPLICATE VARIABLE ERROR

Trying to DEFINE a variable name twice, or trying to name a procedure and a variable by the same name will cause this error.

6 = DUPLICATE PROC/FUNC ERROR

Trying to define a **PROCEDURE** or **FUNCTION** name twice, or defining one with the name of a variable will cause this error.

7 = SYMBOL TABLE FULL ERROR

There is insufficient room to define any more variables, procedures, functions or labels.

8 = UNDEFINED SYMBOL ERROR

Reference was made to a variable or procedure which does not exist. Spelling errors are the main cause of this error message.

9 = UNEXPECTED END OF FILE ERROR

If the closing statement for a block (**ENDIF**, **ENDPROC**, **ENDFUNC**, **ENDWHILE**) is left out, NADOL will run out of text trying to find it.

11 = VALUE RANGE ERROR

Some likely causes:

Assigning a value >255 to a BYTE variable.
Hires or lores coordinates out of range.
GOTOXY parameters out of range.

13 = NESTED LABEL ERROR

Labels may not be defined within **IF-ELSE-ENDIF** or **WHILE-ENDWHILE** blocks.

14 = SUBSCRIPT ERROR

A variable's subscript could not be understood. Possibly missing the ']' after the subscript.

15 = NO BEGIN ERROR

An **ENDPROC** or **ENDFUNC** statement was encountered without the corresponding **PROCEDURE** or **FUNCTION** statement.

16 = WRONG TYPE OF PARAMETER ERROR

A constant was specified in place of a variable name for a procedure which requires a variable name for one of its parameters.

17 = READ ONLY ERROR

Some of the predefined variables may be referenced but may not be modified.

19 = IMMEDIATE ONLY ERROR

The **LIST**, **EDIT**, and **RUN** commands may not be executed from within a program.

20 = NO LANGUAGE CARD ERROR

An attempt was made to perform a data transfer to/from a language card when none was present.

21 = NO AUXILIARY MEMORY ERROR

An attempt was made to perform a data transfer to/from auxiliary memory when none was present.

22 = IF/ENDIF MISMATCH ERROR

The IF-ENDIF blocks are out of balance in a program.

23 = WHILE/ENDWHILE MISMATCH ERROR

The WHILE-ENDWHILE blocks are out of balance in a program.

24 = PROGRAM TO LARGE ERROR

An attempt was made to load a file which is to large for the program area. This is normally caused by attempting to load a non-program file without the **AT** option.

25 = I/O ERROR

A disk error occurred while trying to access the disk. Normally caused by an open disk drive door or a bad diskette.

26 = DISK FULL ERROR

All of the available space on a data diskette has been used and no additional data can be stored on it.

28 = FILE NOT FOUND ERROR

The specified file name could not be located on the current work disk.

29 = NO APPLESOFT ERROR

Hires graphics were attempted without Applesoft BASIC in ROM.

Appendix B

Decimal / Hexadecimal / ASCII conversion chart

DEC	HEX	ASCII	DEC	HEX	ASCII	DEC	HEX	ASCII
0	0	NUL	43	2B	+	86	56	V
1	1	SOH	44	2C	,	87	57	W
2	2	STX	45	2D	-	88	58	X
3	3	ETX	46	2E	.	89	59	Y
4	4	EOT	47	2F	/	90	5A	Z
5	5	ENQ	48	30	0	91	5B	[
6	6	ACK	49	31	1	92	5C	\
7	7	BEL	50	32	2	93	5D	]
8	8	BS	51	33	3	94	5E	^
9	9	HT	52	34	4	95	5F	_
10	A	LF	53	35	5	96	60	`
11	B	VT	54	36	6	97	61	a
12	C	FF	55	37	7	98	62	b
13	D	CR	56	38	8	99	63	c
14	E	SO	57	39	9	100	64	d
15	F	SI	58	3A	:	101	65	e
16	10	DLE	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6A	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20		75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	'	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	54	T	127	7F	
42	2A	*	85	55	U			

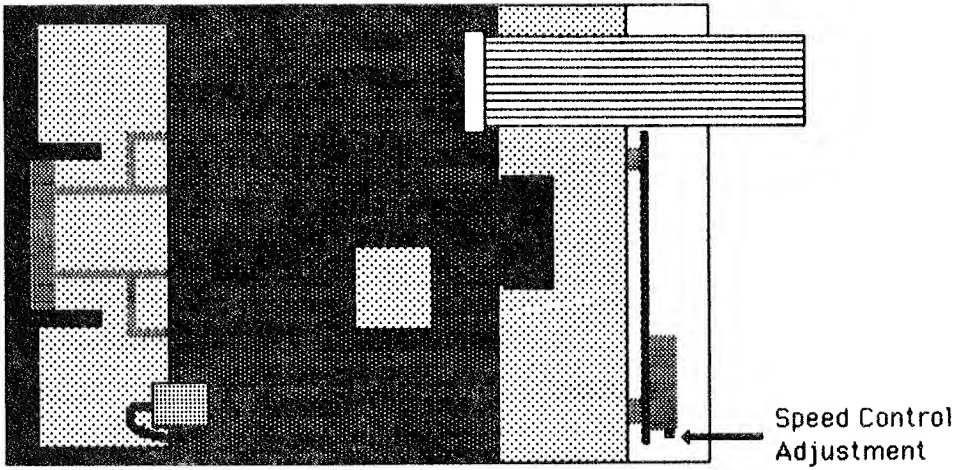
DEC	HEX	ASCII	DEC	HEX	ASCII	DEC	HEX	ASCII
128	80	NUL	171	AB	+	214	D6	V
129	81	SOH	172	AC	,	215	D7	W
130	82	STX	173	AD	-	216	D8	X
131	83	ETX	174	AE	.	217	D9	Y
132	84	EOT	175	AF	/	218	DA	Z
133	85	ENQ	176	B0	0	219	DB	[
134	86	ACK	177	B1	1	220	DC	\
135	87	BEL	178	B2	2	221	DD	]
136	88	BS	179	B3	3	222	DE	^
137	89	HT	180	B4	4	223	DF	_
138	8A	LF	181	B5	5	224	E0	`
139	8B	VT	182	B6	6	225	E1	a
140	8C	FF	183	B7	7	226	E2	b
141	8D	CR	184	B8	8	227	E3	c
142	8E	SO	185	B9	9	228	E4	d
143	8F	SI	186	BA	:	229	E5	e
144	90	DLE	187	BB	;	230	E6	f
145	91	DC1	188	BC	<	231	E7	g
146	92	DC2	189	BD	=	232	E8	h
147	93	DC3	190	BE	>	233	E9	i
148	94	DC4	191	BF	?	234	EA	j
149	95	NAK	192	C0	@	235	EB	k
150	96	SYN	193	C1	A	236	EC	l
151	97	ETB	194	C2	B	237	ED	m
152	98	CAN	195	C3	C	238	EE	n
153	99	EM	196	C4	D	239	EF	o
154	9A	SUB	197	C5	E	240	F0	p
155	9B	ESC	198	C6	F	241	F1	q
156	9C	FS	199	C7	G	242	F2	r
157	9D	GS	200	C8	H	243	F3	s
158	9E	RS	201	C9	I	244	F4	t
159	9F	US	202	CA	J	245	F5	u
160	A0		203	CB	K	246	F6	v
161	A1	!	204	CC	L	247	F7	w
162	A2	"	205	CD	M	248	F8	x
163	A3	#	206	CE	N	249	F9	y
164	A4	$	207	CF	O	250	FA	z
165	A5	%	208	D0	P	251	FB	{
166	A6	&	209	D1	Q	252	FC	|
167	A7	'	210	D2	R	253	FD	}
168	A8	(	211	D3	S	254	FE	~
169	A9	)	212	D4	T	255	FF	
170	AA	*	213	D5	U			

Appendix C

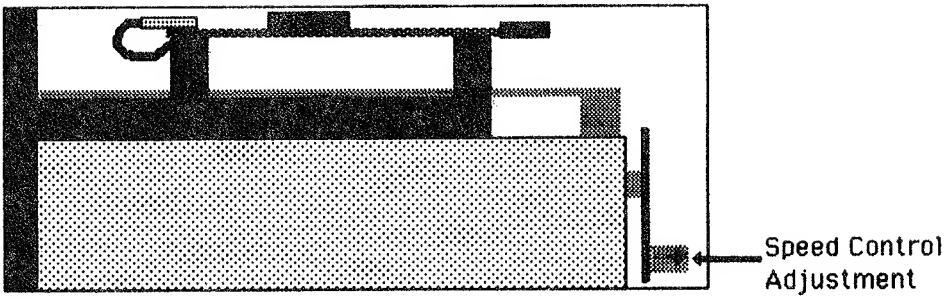
Adjusting Disk Drive Speed

Disk Speed Adjustment

Top View



Side View

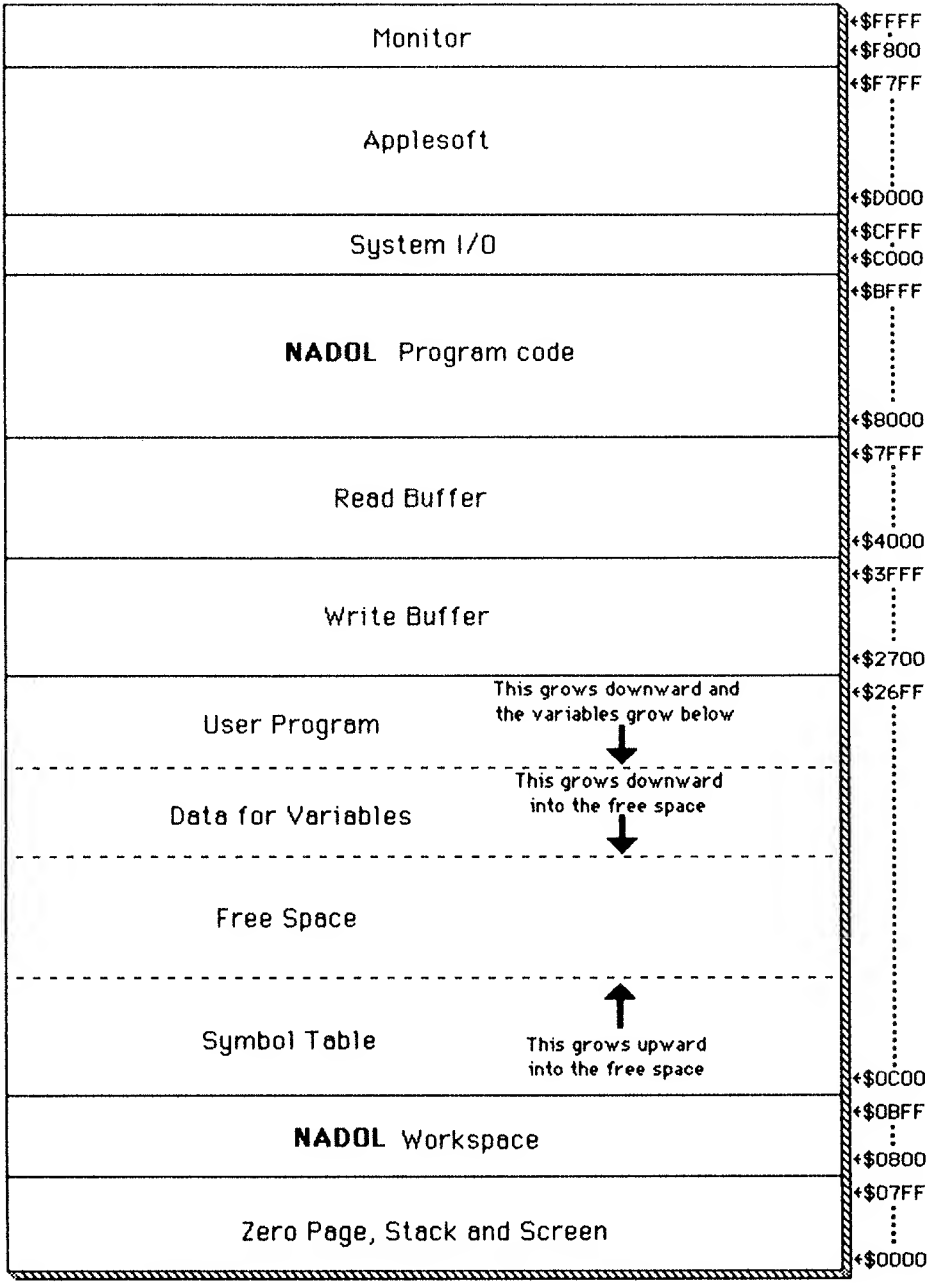


Use the disk speed test option in NAIII. While watching the indicator on the screen, slowly change the speed adjustment. When the desired speed is reached, let the drive stabilize for a moment to verify the correct speed, then the test is complete.

Appendix D

NADOL Memory Map

Memory Map


Monitor
$FFFF
$F800
Applesoft
$F7FF
$D000
System I/O
$CFFF
$C000
NADOL Program code
$BFFF
$8000
Read Buffer
$7FFF
$4000
Write Buffer
$3FFF
$2700
User Program
This grows downward and the variables grow below
$26FF
Data for Variables
This grows downward into the free space
Free Space
Symbol Table
This grows upward into the free space
$0C00
NADOL Workspace
$0BFF
$0800
Zero Page, Stack and Screen
$07FF
$0000

Appendix E

Quick Reference Guide

```
IF expression
    .
    ( statements executed on true )
    .
( ELSE )
    .
    ( statements executed on false )
    .
ENDIF
IN# expression
INIT name
INPUT(«name»,max,«count»)
INVERSE
LABEL name
LCMOVE(«mem•address»,lc•address,length,direction)
variable= LENGTH(name)
LIST
LOAD filename ( AT «address» )
LORES
variable= LSCRN(x,y)
MAKE(«address»,length,start,num•zeroes,bit•length)
MASK(«start»,length,or_value,and_value)
NEW
variable= NOT(expression)
NORMAL
PACK «name» WITH "text"
variable= PDL(expression)
PLOT(x,y)
PR# expression
PRBLOCK(«start»,length,label,digs,format,num1,num2,space)
PRINT ( expression ) (,) (;) (expression) (,) (;)...
PRINTBYTE (expression) (,) (;) (expression) (,) (;)...
PRINTHEX (expression) (,) (;) (expression) (,) (;)...
PROCEDURE name
RECAL(slot,drive)
RENAME oldname,newname
RBLOCK(«address»,«block»,«count»,slot,drive,«error»)
RESULT= expression
RSECT(«address»,«track»,half,«sector»,«count»,slot,drive,«error»)
RTRACK(«address»,track,half,slot,drive)
RUN
SAVE filename (AT «address»,length)
```

Quick Reference to Procedures and Functions

The following is a condensed list of the proper syntax for all of the built-in statements. It is provided as a quick-reference quide, section 8 should be consulted for full details on all of the statements.

AUXMOVE(«apple•addr»,aux•addr,length,direction)
BEEP(tone,time)
CALL(«address»,«accumulator»,«x•register»,«y•register»,«status»)
CATALOG
CLEAR
CLREOL
CLREOP
COLOR= expression
CONVERT(«source»,«destination»,type,size,«count1»,«count2»)
COPY(«source»,«destination»,length)
(DEFINE) type ([n]) name (,name) ....
DELAY(expression)
DELETE filename
DISASM(«start»,label,lines,«offset»)
DISPLAY(«start»,length)
EDIT
FILL(«start»,length,value)
FIND(«start»,length1,«pattern»,length2,7•flag,wild•flag,«offset»)
FLASH
FORMAT(first,last,volume,«interleave»,nsect,slot,drive,error)
variable= FREE
FUNCTION name
GOTO labelname
GOTOXY(x,y)
HCOLOR= expression
HEXPACK «name» WITH "text" (, checksum)
HIRES
HLINE(x1,y1,x2)
HOME
HPLOT (x,y) (TO x,y)....
variable= HSCRN(x,y)

```
SETFORMAT(type,«address•header»,«data•header»,«interleave»)
variable- SIZEOF(filename)
STOP
TEXT
VLINE(x1,y1,y2)
WBLOCK(«address»,«block»,«count»,slot,drive,«error»)
WHILE expression
   .
   .
   (executable statements)
   .
   .
ENDWHILE
WORKDRIVE slot,drive
WSECT(«address»,«track»,half,«sector»,«count»,slot,drive,«error»)
```